


# Desarrollo de una inteligencia artificial para un juego de baloncesto

**Isaac Clerencia Pérez**

PROYECTO FIN DE CARRERA

**Septiembre de 2010**

Director: Diego Gutierrez
Departamento de Informática e Ingeniería de Sistemas
Ingeniería en Informática
Centro Politécnico Superior, Universidad de Zaragoza

---

Curso 2009-2010

Desarrollo de una inteligencia artificial para un juego de baloncesto

# RESUMEN

El trabajo realizado en este proyecto ha sido el desarrollo de un juego de baloncesto con una inteligencia artificial para que una persona pueda jugar contra ella. El resultado es el primer simulador deportivo 3D bajo una licencia libre.

En primer lugar se hizo un análisis del proyecto en el que se estudió el juego completo con especial énfasis en la inteligencia artificial.

A continuación se realizó un diseño de alto nivel de los principales subsistemas del juego.

En cuanto a la estructura general del juego, se planeó un bucle principal con soporte para diferentes estados de juego, utilizando una pila para mantener los estados actualmente en ejecución, incluyendo menú, partido y pausa. De esta manera el estado encargado de recibir eventos y actualizar los gráficos y otras variables del juego es el que se encuentra en el tope de la pila. Es muy sencillo volver de un estado al anterior pues sólo tenemos que desapilar el tope.

La siguiente parte importante fueron las estructuras de datos para el partido (jugadores, equipos y partido), sus contrapartes gráficas en 3D y la relación entre ellas. Además se programó las partes de simulación física para obtener un comportamiento realista de la pelota cuando rebota contra el suelo o el aro, así como para detectar colisiones entre jugadores o intercepciones de la pelota.

Por último, la parte más importante fue la inteligencia artificial, objetivo principal del proyecto. Una parte fundamental de la inteligencia artificial es la captura de información para poder tomar decisiones adecuadas. No es eficiente que la IA compruebe a cada momento el estado de todo el resto de jugadores, por lo que se desarrolló un sistema de eventos, de tal modo que la inteligencia artificial es notificada de cualquier hecho importante en el partido. Este sistema de eventos usa un patrón *Observer* de tal manera que los objetos que quieren recibir eventos pueden suscribirse a él.

Contando con el sistema de gestión de eventos, se desarrolló una inteligencia artificial jerárquica contando con tres niveles fundamentales a nivel de jugador y un nivel superior de equipo.

Los tres niveles de IA individual culminan en un sistema de acciones de alto nivel en el que es fácil dar instrucciones complejas a los jugadores.

En el sistema de acciones se apoya la IA de equipo, basada en una máquina de estados capaz de reaccionar ante los eventos del partido para elegir el estado adecuado para actuar ante la situación actual del partido.

Para incrementar la flexibilidad de la IA y ponerla al alcance del usuario final se implementó la posibilidad de modificar la IA mediante scripting sin necesidad de recompilar el juego. Es posible añadir o redefinir estados de la IA de equipo escribiendo sencillos ficheros en *Javascript*.

Para ayudar a mejorar la IA se cuenta con un sistema de pruebas que permite definir escenarios y ejecutarlos de manera automática para comprobar el efecto que posibles cambios en la IA producen en el juego.

# Índice general

# Capítulo 1

# Introducción

## 1.1. Objetivo

El objetivo de este proyecto es diseñar e implementar un juego de baloncesto en tres dimensiones que incluya una inteligencia artificial. Este juego será el primero de este tipo en ser publicado bajo una licencia libre.

El entorno para el desarrollo ha sido el sistema operativo Linux. Se ha utilizado C++ como lenguaje principal de implementación y la biblioteca Ogre3D como motor gráfico. El juego es multiplataforma y es posible ejecutarlo en Linux, MacOS X y Windows.

El software desarrollado se distribuye bajo licencia GNU GPL.

## 1.2. Alcance

La complejidad del problema abordado es considerable, especialmente en algunos aspectos como la física del juego y la inteligencia artificial.

Para poder trabajar en el desarrollo de la inteligencia artificial, prioridad del proyecto, ha sido necesario desarrollar anteriormente el resto de partes necesarias del juego, como la física o el sistema de eventos.

## 1.3. Trabajo previo

En el desarrollo de este proyecto se aprovechan una serie de bibliotecas y herramientas preexistentes que han hecho posible la implementación en un periodo de tiempo razonable. Entre las más importantes se cuentan:

- Distribuciones de GNU/Linux Debian/Ubuntu
- Motor 3D OGRE

- Biblioteca de simulación de dinámica de solidos rígidos ODE
- Biblioteca de interfaz gráfica CEGUI
- Bibliotecas C++ Boost
- Toolkit Qt para tipos abstractos de datos, acceso a bases de datos y red
- Lenguaje de programación C++

Además se utilizan otras bibliotecas de C++ menos importantes para facilitar tareas como la gestión de argumentos en la línea de órdenes.

## 1.4. Motivación del proyecto

El proyecto se planteó debido a un interés personal en el desarrollo de videojuegos. El diseño y programación de los mismos plantea un gran número de interesantes retos tanto a nivel de arquitectura del programa como a nivel de física e inteligencia artificial.

## 1.5. Líneas generales

### 1.5.1. Estrategia

El proyecto se ha dividido en tres fases principales:

**Investigación:** durante esta fase se realizó el estudio de los distintos campos que abarca este proyecto como son la inteligencia artificial en videojuegos, la física, reglas del juego, etc. Los resultados obtenidos se reflejan en los anexos correspondientes. El tiempo a emplear en esta primera fase se fijó en un mes.

**Análisis y diseño:** se realizó el análisis del proyecto y un diseño de alto nivel para la estructura general del juego. Esta fase duro entorno a un mes.

**Desarrollo:** implementación del juego. Se planificó inicialmente en tres meses pero el tiempo final fue de cuatro meses.

En la tabla 1.1 se expone el cronograma estimado.

| **Id** | **Paquete de trabajo** | **Inicio** | **Fin** |
|---|---|---|---|
| 1 | Investigación | $t_0$ | $t_0 + 1m$ |
| 2.1 | Desarrollo: análisis y diseño | $t_0 + 1m$ | $t_0 + 2m$ |
| 2.2 | Desarrollo: implementación | $t_0 + 2m$ | $t_0 + 5m$ |

Tabla 1.1: Cronograma estimado

En la tabla 1.2 se expone el cronograma real.

| Id | Paquete de trabajo | Inicio | Fin |
|---|---|---|---|
| 1 | Investigación | $t_0$ | $t_0 + 1m$ |
| 2.1 | Desarrollo: análisis y diseño | $t_0 + 1m$ | $t_0 + 2m$ |
| 2.2 | Desarrollo: implementación | $t_0 + 2m$ | $t_0 + 6m$ |

Tabla 1.2: Cronograma real

### 1.5.2. Métodos y técnicas

El empleo de LaTeX como sistema de documentación permitió una integración sencilla de capítulos escritos de forma independiente. Así como un fácil mantenimiento de la bibliografía.

La metodología de ingeniería del software empleada ha sido la programación extrema (Extreme Programming).

La programación extrema es una de las metodologías para el desarrollo ágil de software. Está especialmente indicada para sistemas donde los requisitos pueden cambiar, como es el caso de este proyecto, cuyos objetivos no están claramente definidos aparte del general, creación de una AI.

Uno de los principios de la programación extrema es el *feedback*. En el caso de los proyectos tradicionales, el *feedback* se obtiene a través del contacto con el cliente. Si ésto ocurre frecuentemente, en iteraciones pequeñas, la respuesta se obtiene antes y se puede aprender y realizar cambios de forma más rápida. En el caso de este proyecto, el papel de cliente lo han asumido terceras personas que han probado el juego y que han proporcionado información de gran utilidad acerca de posibles mejoras o problemas existentes.

### 1.5.3. Herramientas

#### Desarrollo

Las tareas de desarrollo han constado principalmente de codificación. También se han utilizado herramientas adicionales para la gestión del proyecto o el control de versiones.

Las herramientas utilizadas para todo ello se describen a continuación:

- Vim: Potente editor de texto.
- Blender: Herramienta de modelado 3D
- Subversion: Herramienta de control concurrente de versiones.
- Trac: Sistema de gestión de proyectos software.

#### Documentación

Las herramientas utilizadas para la edición y generación de toda la documentación han sido las siguientes:

- Vim: Potente editor de texto.
- LaTeX: Lenguaje de marcado para la creación de documentos, formado por un conjunto de macros de TeX. Se ha utilizado para la generación de la presente documentación.
- inkscape: Programa de dibujo vectorial
- Dia: Editor de diagramas

## 1.6. Convenciones adoptadas

#### Representación de nombres de fichero y comandos

- Los nombres de fichero a los que se haga referencia en el texto aparecerán siempre en letra *cursiva*.
- Los nombres correspondientes a órdenes del sistema u otros programas aparecerán en letra **negrita**.

#### Código fuente con realzado de sintaxis

Los fragmentos de código fuente y ficheros de especificación que tengan un formato determinado aparecerán en un recuadro gris como se muestra a continuación:

```
int main() {
    printf("Hello, world");
}
```

#### Contenido de ficheros y salida de programas

La salida de programas o contenido de ficheros que no tengan una determinada sintaxis aparecerá en fuente monoespaciada con el siguiente formato:

```
Render System=OpenGL Rendering Subsystem

[OpenGL Rendering Subsystem]
FSAA=0
Full Screen=No
RTT Preferred Mode=PBuffer
Video Mode=800 x 600
```

## 1.7. Estructura de este documento

La documentación del proyecto consta de la memoria principal y una serie de anexos, acompañados de la bibliografía.

La memoria está organizada en cuatro capítulos: introducción, trabajo desarrollado, resultados y conclusiones. En el primero de ellos se describe el objetivo del proyecto, su alcance, el trabajo previo, el contexto, las líneas generales y una breve reseña de las convenciones utilizadas. El capítulo sobre el trabajo desarrollado ofrece una visión general sobre cada una de las principales tareas realizadas y sistemas implementados. Finalmente, el capítulo que contiene las conclusiones está dividido en tres secciones: descripción resumida, valoración crítica e incidencias.

Por otro lado, los anexos incluidos son los siguientes:

**A. Diseño:** Este anexo ofrece información sobre el diseño del juego, sin entrar en detalles en la física y la inteligencia artificial.

**B. Física:** Este anexeo ofrece detalles sobre la resolución de los problemas relacionados con la simulación física en el juego.

**C. Inteligencia artificial:** Este anexo contiene ́la descripción del diseño de la inteligencia artificial para el juego.

**D. Licencia:** Este anexo contiene el texto oficial de la GNU General Public License, bajo la cual se licencia todo el software de este proyecto.

# Capítulo 2

# Trabajo desarrollado

En este capítulo se exponen las principales tareas desarrolladas para la elaboración del presente proyecto.

El objetivo principal del proyecto era la creación de una inteligencia artificial para un juego de baloncesto, pero para empezar a desarrollar está inteligencia artificial hace falta una gran cantidad de infraestructura.

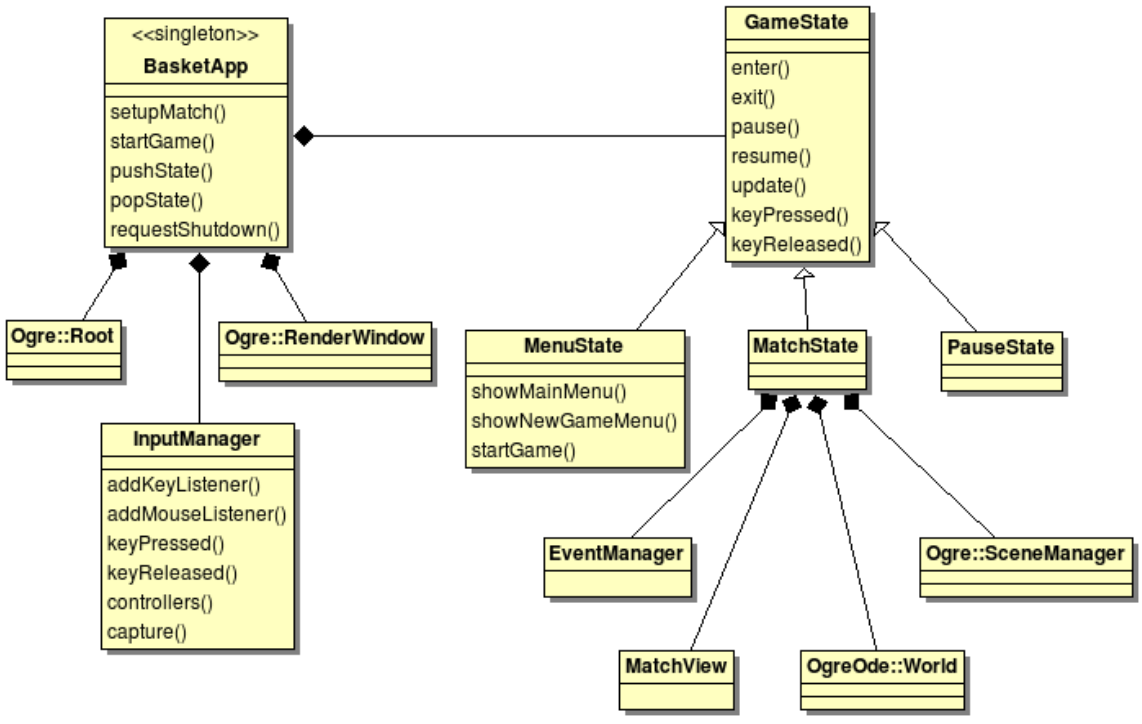



Figura 2.1: Diagrama de clases principal

En primer lugar, como en todos los videojuegos [4], la mayor parte del tiempo de ejecución se va a dedicar al bucle principal. Este bucle principal recibe la entrada de los controladores,

calcula el nuevo estado basándose en esta entrada y en la inteligencia artificial y actualiza la visualización en pantalla.

Este bucle principal se encuentra en el método **BasketApp::startGame()**. Para dar soporte al bucle principal existe un sistema de estados en los que el juego puede estar. Estos estados son *menú*, *partido* y *pausa*, cada uno de ellos implementado en una clase distinta. El bucle principal tiene una pila con los estados cargados y se limita a enviar la entrada de los controladores al estado actual y a invocar a su función de actualización. Las clases fundamentales del juego se muestran en la figura A.1.

Una vez diseñado el bucle principal y la gestión de estados, el siguiente paso necesario fue la representación visual del partido. Utilizando un motor 3D se creó una visualización simple del partido, creando modelos en tres dimensiones para la pista, las canastas y jugadores básicos formados por un cuerpo cilíndrico y dos brazos.

## 2.1. Física

La física es una parte fundamental de la simulación de un deporte en un programa informático si queremos alcanzar un grado mínimo de realismo. En este proyecto los principales problemas de este tipo que se deben resolver están relacionados con el movimiento del balón de baloncesto y las colisiones entre éste, los jugadores y la cancha.

Para realizar estos cálculos físicos se utiliza un motor de física, es decir, una biblioteca que simula los modelos físicos de Newton, teniendo en cuenta variables como la velocidad, masa y fricción de los cuerpos, de tal manera que es posible calcular de manera aproximada como se comportarían los objetos de nuestro juego en el mundo real.

En primer lugar debemos distinguir entre los motores de alta precisión y los de tiempo real. Los primeros, como su nombre indica, tienen como objetivo alcanzar un alto grado de precisión, mientras que el tiempo necesario para llegar a ese resultado no es tan importante. Este tipo de motores se utilizan sobre todo para realizar calculos científicos. Por otra parte, en un juego o en otras aplicaciones interactivas, lo más importante son las estrictas restricciones temporales, por lo que el motor debe de ser capaz de calcular los resultados físicos en tiempo real, es decir, tan rápido como sucederían en realidad. En este tipo de motores la precisión del resultado final pasa a un segundo plano, bastando una aproximación lo suficientemente buena.

Los motores de física suelen tener dos partes diferenciadas: un simulador de dinámica de sólidos rígidos y un sistema de detección de colisiones. El primero determina como actúan las fuerzas sobre los objetos del sistema. El segundo calcula cuando han tenido lugar colisiones entre los objetos y añade fuerzas al sistema para simular estas colisiones.

### 2.1.1. Elección de un motor de física

Un motor de física en tiempo real es un programa relativamente complejo, por lo que no era una opción desarrollar uno para este proyecto, ya suficientemente extenso. Por tanto, se ha decidido utilizar un motor de física ya existente, por lo que durante la fase de investigación se analizaron algunos de ellos.

Los requisitos que debían cumplir eran los siguientes:

- Software libre
- Multiplaforma, al menos disponible para Unix/Linux, Mac OS X y Windows
- Desarrollado y mantenido activamente
- Documentación adecuada
- Soporte para, al menos, planos, esferas, cilindros y mallas de triangulos

Las tres primeras condiciones limitan la elección a PAL [1], Bullet [2] y ODE [3].

### PAL

PAL (Physics Abstracion Layer) proporciona una interfaz unificada para varios motores de física. Esto permite utilizar varios motores en una única aplicación. Además de esta capa de abstracción, también proporciona una arquitectura de plug-ins para el sistema de física.

Las principales ventajas de PAL son su facilidad de uso y la flexibilidad que aporta el hecho de poder usar diferentes motores de física. De este modo se puede empezar el desarrollo con un motor de física libre y cambiar a uno comercial en caso de ser necesario sin tener que realizar cambios en el código. Otras ventajas son la posibilidad de utilizar diferentes motores de física en diferentes plataformas y la seguridad de poder cambiar a otro motor de física si, por ejemplo, el que utilizamos deja de ser desarrollado o tener soporte.

### Bullet

Bullet es una biblioteca libre de detección de colisiones y dinámica de sólidos rígidos para videojuegos.

Bullet tiene un gran rendimiento en la detección de colisiones entre cuerpos muy complejos. Es un motor muy reciente, lo que le proporciona algunas ventajas, como una moderna arquitectura que permite su uso de una manera sencilla en programas multihilo, especialmente importante para el desarrollo para consolas de videojuegos. Cuenta con una comunidad activa de desarrollo que sigue añadiendo características al motor.

### ODE

ODE es una biblioteca de alto rendimiento para la simulación de dinámica de sólidos rígidos. Tiene todas las características básicas de un motor de física y es estable, maduro y multiplataforma. Tanto la documentación como el soporte son excelentes. ODE es utilizado en multitud de proYectos, tanto libres como propietarios. Ogre cuenta con un plug-in llamado OgreODE que integra de manera muy sencilla ODE con Ogre.

[1] http://www.adrianboeing.com/pal/
[2] http://www.bulletphysics.com/Bullet/wordpress/
[3] http://www.ode.org/

#### Decisión

Aunque la simulación física es una parte muy importante del juego, el uso que se realiza del motor es bastante básico. El código relacionado con el motor de física es relativamente pequeño por lo que un eventual cambio de motor no supone un esfuerzo significativo. Debido a esto no se consideró la flexibilidad de PAL como una ventaja importante.

Bullet es un motor físico con un gran potencial, bajo intenso desarrollo y con características muy interesantes, pero todas las características necesarias para el juego también son proporcionadas por ODE, considerado generalmente el motor más estable, probado y con mejor soporte.

Si también se tiene en cuenta la existencia del plug-in OgreODE que facilita tremendamente la integración del motor físico con el motor gráfico, ODE es definitivamente la opción más razonable para el juego.

### 2.1.2. Open Dynamics Engine

Las principales características de ODE son las siguientes:

#### Simulación de estructuras articuladas

ODE es capaz de simular estructuras de sólidos rígidos artículados, unidos por diferentes juntas o articulaciones. Ejemplos de tales objetos pueden ser vehículos terrestres o cualquier tipo de criatura con piernas.

#### Apto para simulaciones interactivas y en tiempo real

ODE ha sido diseñado de manera que tiene un gran rendimiento y estabilidad, por lo que es posible su utilización en simulaciones interactivas y en tiempo real. El usuario tiene completa libertad para cambiar la estructura del sistema incluso cuando la simulación está en ejecución. El integrador de ODE es altamente estable, por lo que los pequeños errores durante la simulación se mantienen bajo control. ODE enfatiza la velocidad y la estabilidad sobre la precisión física.

#### Sistema de detección de colisiones

ODE dispone de un sistema de detección de colisiones integrado, con capacidad para calcular colisiones entre un amplio número de primitivas, como esferas, cajas, planos y mallas de triángulos.

### 2.1.3. Utilización de ODE en un proyecto con Ogre

Gracias al *wrapper* OgreODE[4] es muy sencillo utilizar ODE en un proyecto que utiliza el motor 3D Ogre.

En primer lugar debemos crear un *mundo* en el que habitaran nuestros sólidos y establecer las variables físicas que se deberán aplicar en este mundo:

[4] http://www.ogre3d.org/wiki/index.php/OgreODE

```
mWorld = new OgreOde::World(mSceneMgr);
mWorld->setGravity(Ogre::Vector3(0,-9.80665,0));
mWorld->setAutoSleep(true);
mWorld->setAutoSleepAverageSamplesCount(10);
```

La principal variable que debemos tener en cuenta es la gravedad. En el caso de que una unidad de Ogre se corresponda a un metro, como en el ejemplo, debemos usar -9.8 si queremos una gravedad similar a la existente en la tierra. Este valor debe ser adaptado si usamos otra escala.

Las funciones de *auto sleep* hacen que el motor de física no haga cálculos innecesarios para objetos que se encuentran en estado de reposo.

Una vez creado y configurado nuestro mundo, debemos establecer un tamaño y una masa para cada objeto gestionado por Ogre que queremos que forme parte de nuestra simulación físicas. Suponiendo que tenemos un nodo de Ogre llamado *node* y queremos representarlo físicamente como una caja:

```
body = new OgreOde::Body(world);
node->attachObject(body);
Vector3 size(10.0,10.0,10.0);
OgreOde::BoxMass mass(0.5,size);
mass.setDensity(5.0,size);
geom = (OgreOde::Geometry*)new OgreOde::BoxGeometry(size, world, space);
mode->setScale(size.x * 0.1,size.y * 0.1,size.z * 0.1);
body->setMass(mass);
geom->setBody(body);
entity->setUserObject(geom);
```

En primer lugar creamos un nuevo cuerpo de ODE utilizando el objecto *OgreOde::Body*. A continuación asociamos este cuerpo al nodo de Ogre. El siguiente paso es definir una masa y una geometría para nuestro cuerpo.

Una vez que tenemos un sólido con una masa asociado al nodo, el motor físico está preparado para actualizar su estado, para lo que simplemente debemos añadir una llamada al *stepper* de OgreODE en nuestro bucle principal:

```
if (mStepper->step(timeSinceLastFrame))
{
    mWorld->synchronise();
}
```

El *stepper* se encarga de actualizar la posición, velocidad y demás variables físicas para cada uno de los objetos que forman parte de la simulación física.

En cuanto a la detección de colisiones, OgreOde proporciona una clase llamada *CollisionListener* a partir de la cual podemos crear una subclase que implemente el método *collision* y puede ser registrada para recibir las notificaciones de todas las colisiones.

Una vez que se ha detectado una colisión podemos obtener información sobre los objetos que intervienen y decidir como actuar basándonos en ella, por ejemplo, decidiendo ignorarla si ese tipo de objetos pueden atravesarse.

## 2.2. Inteligencia artificial

Contando con el bucle principal, la visualización gráfica y el motor de física fue posible empezar a implementar la inteligencia artificial. El objetivo era tener una inteligencia artificial que ofreciera un juego básico y a la vez fuera ampliable mediante un lenguaje de script de manera sencilla, de tal manera que usuarios avanzados del juego pudieran añadir nuevas jugadas a la inteligencia artificial.

### 2.2.1. Sistema de eventos

La inteligencia artificial necesita datos para poder tomar decisiones y todos esos datos le llegan en forma de eventos. Se creó un gestor de eventos que es capaz de recibir eventos de todas las partes de la aplicación y distribuirlos a los observadores registrados en el mismo, así como escribirlos en un registro para ayudar a la depuración de la inteligencia artificial. En la figura C.2 podemos ver la estructura general del sistema de eventos.

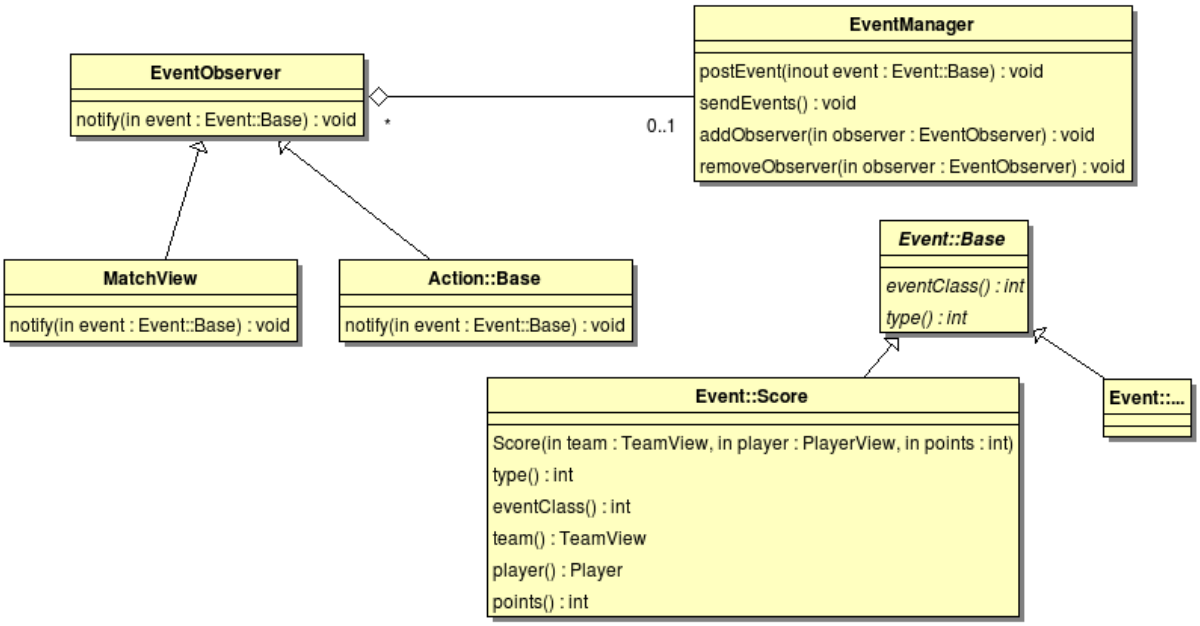



Figura 2.2: Diagrama del sistema de eventos

Los eventos incluyen cualquier circunstancia del partido que una inteligencia artificial puede querer tener en cuenta. Éstas circunstancias se pueden dividir en varias categorías.

En primer lugar tenemos las acciones realizadas por los jugadores como un pase, un tiro a canasta o incluso la preparación de un tiro a canasta, de tal manera que la inteligencia artificial pueda responder intentando taponar.

En segundo lugar tenemos eventos arbitrales, como el pítido de inicio de partido, el de final de posesión o la señalización de una falta.

Por último, tenemos otros eventos que no encajan en las anteriores categorías, como por ejemplo, un evento que se lanza cada segundo de tal manera que la inteligencia artificial tenga una oportunidad de ejecutarse cada segundo aunque no haya eventos en el partido, y otro evento que indica que la acción que un jugador estaba llevando a cabo se ha terminado. Usando este último evento es posible definir jugadas complicadas, como por ejemplo indicar a un jugador que tiene que esperar a que otro compañero de equipo vaya a una posición para realizar un pase.

### 2.2.2. Acciones

La inteligencia artificial usa un sistema de estados para saber cual es el objetivo actual del equipo. Este objetivo puede ser defender, subir la pelota a la canasta contraria o intentar anotar. Los eventos pueden disparar transiciones entre estos estados (como en el caso de final de posesión o de un robo de balón) o simplemente desencadenar acciones sin cambiar el estado, por ejemplo, intentando interceptar un pase realizado por el equipo contrario.

Para dar instrucciones a los jugadores del equipo, la inteligencia artificial utiliza un sistema jerárquico de acciones. En el nivel más alto tenemos acciones de alto nivel como por ejemplo cubrir a un contrario, interceptar la bola, realizar un tiro a canasta o intentar robar el balón. Estas acciones están compuestas por un conjunto de acciones básicas.

Las acciones básicas son los movimientos más simples posibles y se caracterizan por ser deterministas. Incluyen instrucciones como girar, saltar o tirar a canasta con unos parámetros fijos (fuerza y dirección).

En un principio parece que con estos dos tipos de acciones es suficiente, pero cuando se realiza una simulación pronto se hace obvio un grave problema. Los jugadores se chocan entre ellos, se salen del terreno de juego y realizan acciones similarmente no inteligentes. Si tuvieramos que introducir comprobaciones para estas situaciones en cada acción de la inteligencia artificial de alto nivel, sería increiblemente complicado y lento escribir nuevas acciones. Una acción tan sencilla como moverse a una posición del terreno de juego necesitaría varias comprobaciones.

Para resolver este problema, se introdujeron *steering behaviours* [1, 3], esto es, diferentes modos de comportamiento que un agente puede tener en cuanto a su movimiento. Estos comportamientos pueden incluir cosas como ir a una posición, seguir a un jugador, no salirse del terreno de juego o no chocarse con otros jugadores, y pueden ser combinados entre ellos. Para calcular el movimiento de un jugador simplemente se ejecutan los distintos comportamientos y se suma el resultado de cada uno de ellos. Si, por ejemplo, queremos que un jugador vaya a una determinada posición sin salirse de la pista y sin chocarse con otros jugadores, simplemente tenemos que añadir los comportamientos, *ir a una posición de la pista*, *no salirse del campo* y *no chocarse con otros jugadores*. La mayoría del movimiento será guiado por el primer comportamiento, pero si el jugador va a chocarse con otro jugador el comportamiento de *no chocarse* rectificará la dirección del jugador para evitar la colisión. Si al intentar evitar la colisión el jugador va a salir fuera del terreno de juego, el comportamiento apropiado rectificará su dirección y velocidad.

### 2.2.3. Inteligencia artificial de equipo

Una vez que tenemos las acciones básicas que permiten asignar instrucciones a los jugadores podemos centrarnos en la inteligencia artificial a nivel de equipo.

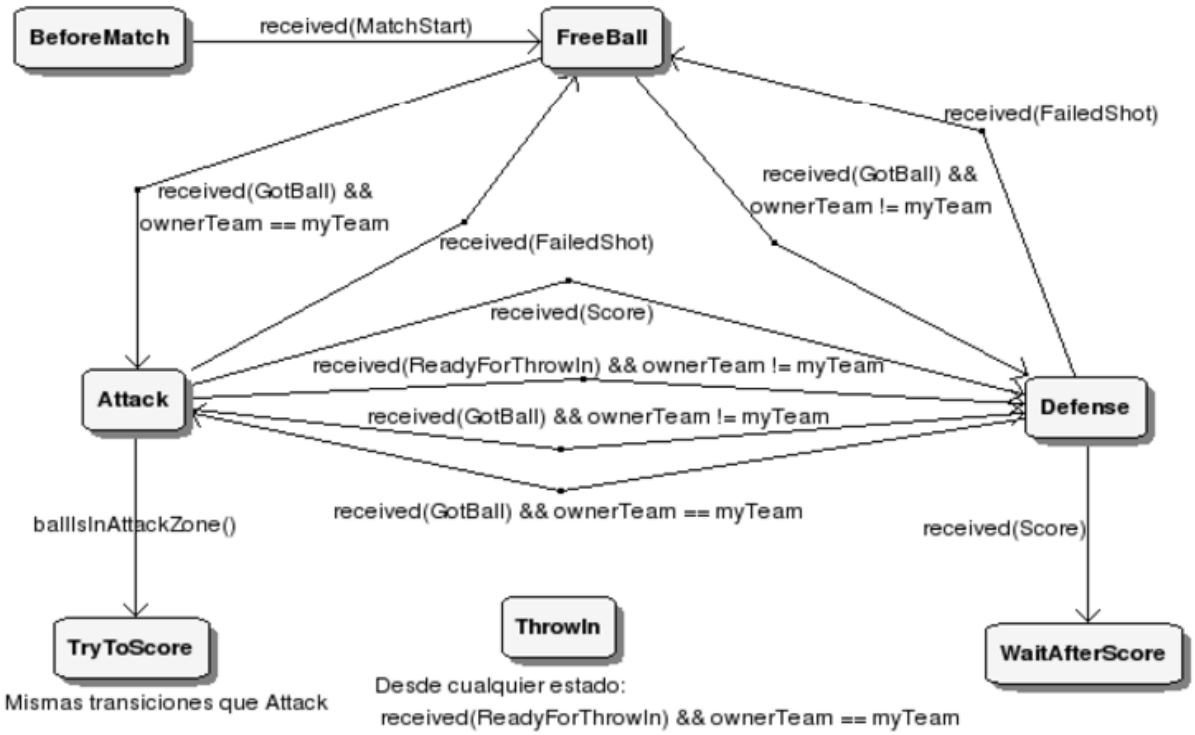



Figura 2.3: Diagrama de estados básico de la IA

La inteligencia artificial de equipo está implementada mediante una máquina de estados, cuyo diagrama de estados básico podemos observar en la figura . Estos estados establecen las acciones a ejecutar por los diferentes jugadores y son notificados de los eventos del partido. Estos eventos recibidos pueden causar una transición a otro estado.

Por ejemplo, el primer estado se llama *BeforeMatch* y su único objetivo es colocar a los jugadores en la posición inicial. Cuando este estado recibe el evento *MatchStart* se pasa al estado *FreeBall*. El objetivo de la inteligencia artificial en este estado es conseguir la posesión de la pelota, y lanzarse al ataque o preparar la defensa dependiendo de la probabilidad estimada de conseguir la posesión. Otros estados básicos son *Attack*, *TryToScore* y *Defend*. Estos tres estados son los que están activos la mayor parte de un partido.

Los estados básicos están implementados en C++ e incluyen métodos para analizar situaciones en el partido, como por ejemplo, estimar la probabilidad de que un pase sea interceptado o las posibilidades de que un tiro a canasta sea exitoso. Estos análisis a menudo incluyen interesantes problemas geométricos.

### 2.2.4. Scripting con Javascript

La IA también incluye una posibilidad de ampliación muy interesante. La mayoría de estados pueden sobreescribirse mediante scripting. También es posible añadir nuevos estados para crear inteligencias artificiales más complejas. Para añadir o modificar estados tan solo es necesario escribir pequeños ficheros en JavaScript, sin necesidad de recompilar el programa.

Esto proporciona una gran herramienta al usuario final para modificar el comportamiento de la IA.

### 2.2.5. Depuración de la IA

Para facilitar la depuración y la mejora de la IA se utilizan tres aproximaciones diferentes.

Por una parte, el programa cuenta con un sistema de registros del juego, basado en log4cpp, que registra de manera jerárquica los eventos que ocurren. De este modo es posible tener registros verbosos que se pueden activar y desactivar fácilmente cuando queremos depurar una parte determinada de la IA.

En segundo lugar, existe un mecanismo llamado *etiquetas de jugadores* que permite a diferentes partes de la IA añadir información de depuración a los jugadores. Estas etiquetas pueden ser activadas durante el partido de tal manera que es posible ver, para cada jugador, que acción están realizando y algunos otros parámetros que explican su comportamiento. Esta información se muestra sobreimpresa acompañando al jugador por el terreno de juego.

Por último, y sobre todo orientado a la mejora de la IA, el tercer método es la ejecución de pruebas. Es posible escribir escenarios de prueba en ficheros XML, conteniendo una configuración inicial del partido, incluyendo las posiciones de los jugadores, quien tiene el balón y cual es el estado de los equipos. Estos escenarios también permiten establecer condiciones de éxito y fracaso, así como un límite de tiempo.

Estos escenarios de prueba, ejecutados un alto número de veces, permiten comprobar el efecto de cambios en la IA en los resultados del juego. mediante la ejecución de los mismos

# Capítulo 3

# Resultados

El resultado del proyecto ha sido un juego de baloncesto en 3D al que pueden jugar dos jugadores o un jugador contra una inteligencia artificial.

El juego consta de un sistema inicial de menús para elegir el tipo de juego deseado, equipos y controladores entre otros.

El jugador controla a su equipo mediante el teclado, pudiendo realizar tiros, lanzamientos a canasta y seleccionar el jugador que desea controlar.

La figura 3.1 contiene una secuencia de partes de fotogramas en la que el usuario está dirigiendo un ataque. En el fotorama número tres podemos observar como su compañero de equipo hace un bloqueo para proporcionar al jugador un tiro claro en el fotograma cuatro, que finalmente se convierte en canasta.

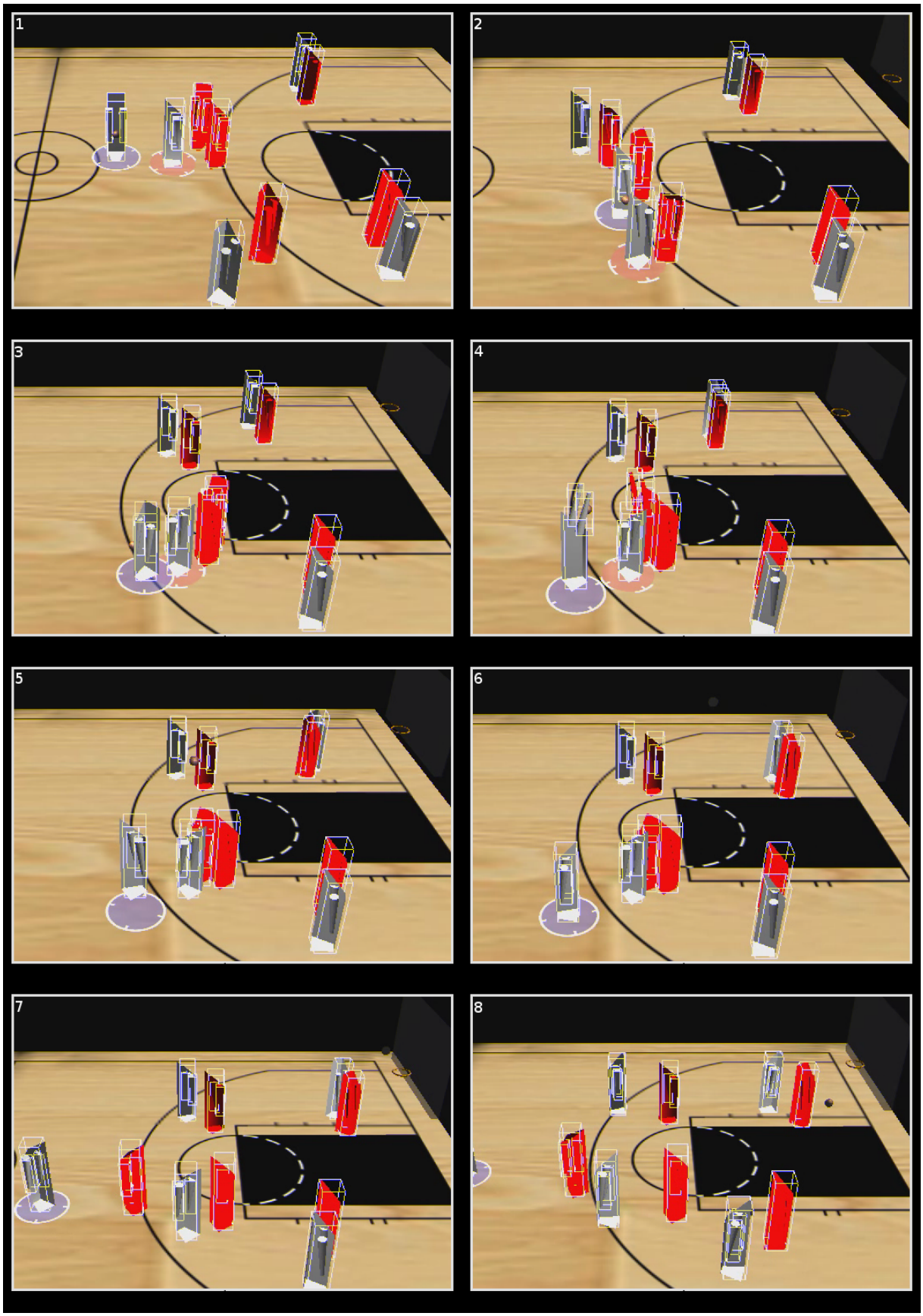



Figura 3.1: Atacando

La inteligencia artificial tiene en cuenta la posición de los jugadores, del balón, el tiempo restante de posesión y es capaz de realizar movimientos que desmarquen a sus jugadores, estimar posibilidades de pase y tiros a canasta, eligiendo las mejores opciones.

La IA no está al nivel de las inteligencias artificiales de juegos profesionales, con gran cantidad de recursos y años de desarrollo, pero resulta un enemigo interesante durante los primeros partidos.

Además, gran parte del esfuerzo realizado se ha centrado, no en que la IA fuera excepcional, sino en que fuera extensible y fácil de mejorar. Para un desarrollador con experiencia en C++ debería ser sencillo mejorar el núcleo de la IA. El proyecto fue ideado desde el primer momento como software libre, por lo que otros desarrolladores están invitados a mejorar el juego. Algunos desarrolladores ya han mostrado su interés en colaborar con el proyecto cuando sea posible.

Por otra parte, otros usuarios menos avanzados pueden intentar añadir nuevas jugadas a la IA utilizando el sistema de scripting.

En la figura 3.2 vemos a la IA realizando un tiro desesperado cuando su posesión está a punto de expirar. Se puede observar el contador de tiempo de partido, el marcador y el reloj de posesión.

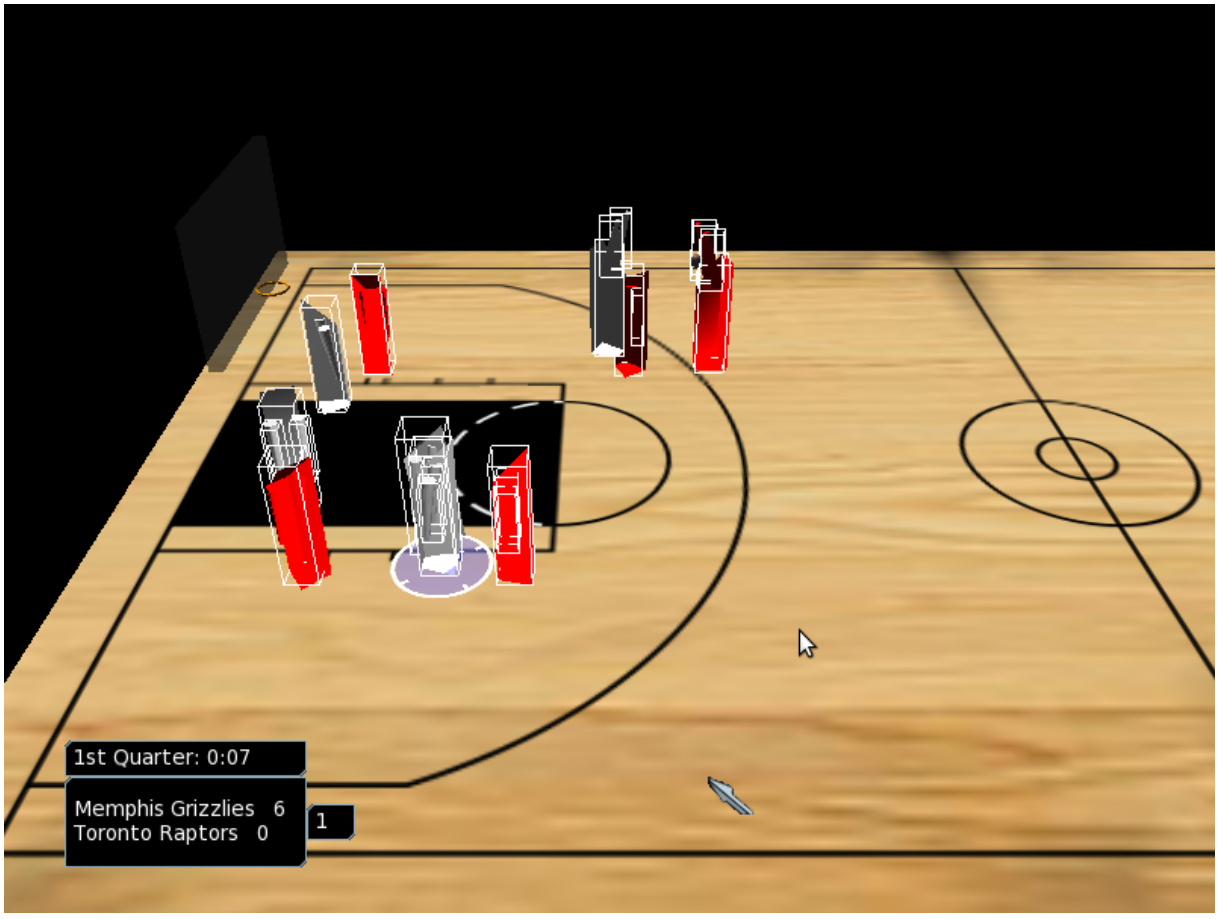



Figura 3.2: Tiro al final de posesión

Como vemos en la figura 3.3, donde también podemos observar las notificaciones de eventos en

el partido, el tiro tuvo éxito.

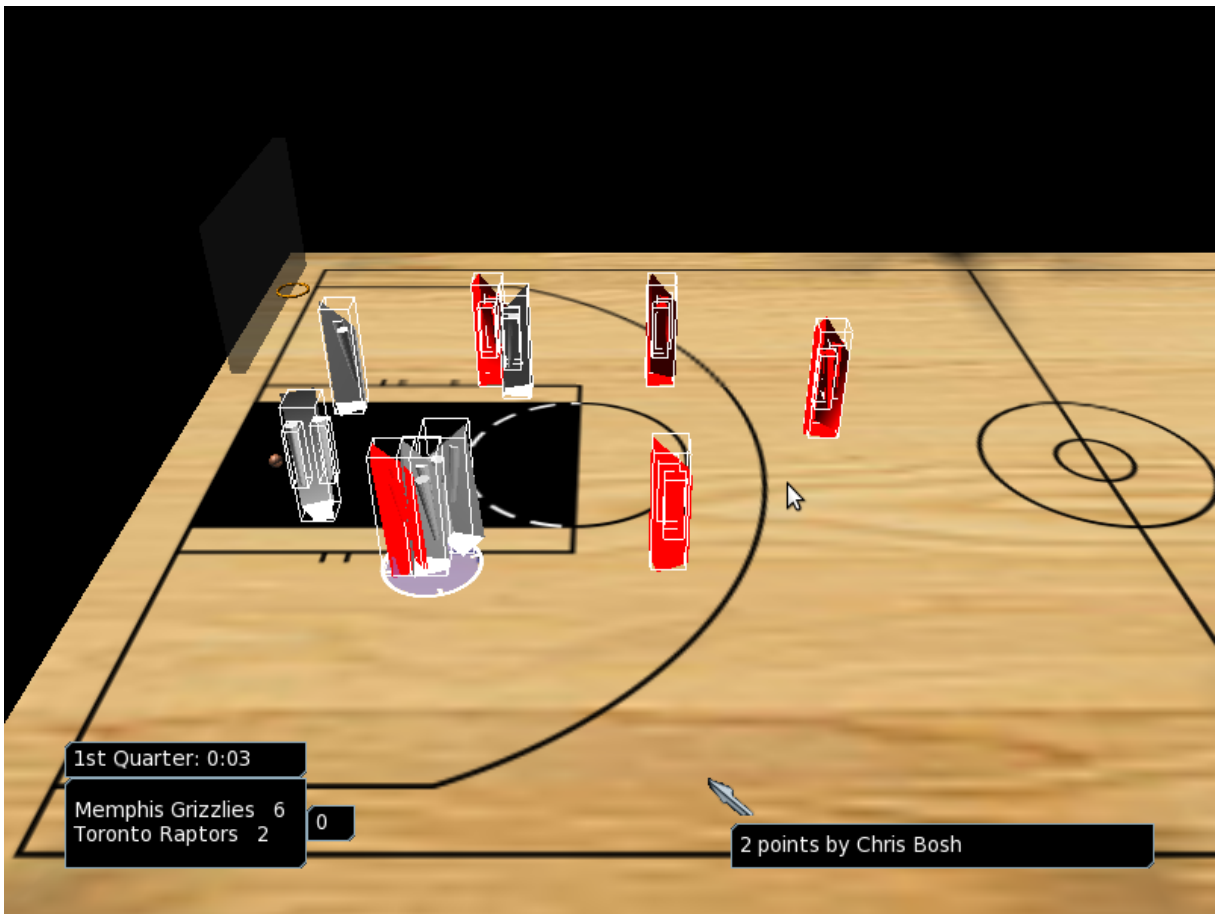



Figura 3.3: Canasta

El proyecto es jugable y resulta divertido durante unos cuantos partidos. Sin embargo la IA todavía es bastante precedible, por lo que resulta fácil aprender algunos trucos que hacen fácil batirla. Sin embargo, la capacidad de mejorar la IA mediante scripting hace que sea muy sencillo añadir nuevas jugadas para hacer el juego más divertido y proporciona una manera alternativa de disfrutar el juego, programando jugadas para la IA.

# Capítulo 4

# Conclusiones

## 4.1. Descripción resumida

Se ha creado un juego de baloncesto en tres dimensiones (In Your Face). El juego incluye una inteligencia artificial para que una persona pueda jugar contra el ordenador, aunque también es posible que dos jugadores humanos jueguen entre ellos.

## 4.2. Valoración crítica

### Aportaciones

En opinión del autor, la principal aportación de este proyecto fin de carrera es la creación del primer simulador deportivo en 3D con una inteligencia artificial bajo una licencia libre. Existen muy buenos simuladores deportivos propietarios, pero hasta ahora no había ninguna implementación libre de una inteligencia artificial de este tipo.

### Cumplimiento de los objetivos

El principal objetivo del proyecto se ha cumplido, la creación de una inteligencia artificial para un juego de baloncesto. Sin embargo, el trabajo necesario para esta tarea ha sido bastante superior al esperado por lo que sólo uno de los objetivos secundarios, la posibilidad de ampliar el juego utilizando un lenguaje de scripting, se ha podido llevar a cabo.

### Trabajo futuro

A pesar de la gran cantidad de trabajo realizado, el proyecto tiene bastantes posibilidades de ampliación. Entre ellas se pueden destacar la posibilidad de juego en red que, debido a la arquitectura de acciones, debería ser sencillo de implementar.

Otras ampliaciones interesantes serían la posibilidad de jugar competiciones completas y la mejora de las estadísticas recogidas durante partidos.

En cuanto al modo de juego, una posibilidad interesante era implementar un interfaz de *tiempo bala*, similar al efecto de la película *Matrix* o de algunos juegos como *Max Payne 2*.

Por supuesto, la inteligencia artificial también puede ser mejorada enormemente. Una de las opciones más interesantes a explorar es modificar la IA para que esté más parametrizada, de tal manera que las constantes que afectan a la inteligencia artificial (como por ejemplo, cuanto riesgo correr en tiros y pases, etc.) sean fácilmente modificables. De este modo es posible crear un sistema de autoentrenamiento de la inteligencia artificial, modificando parámetros y realizando simulaciones hasta obtener mejoras en los resultados.

# Anexo A

# Diseño

En este anexo se ofrece información sobre el diseño del juego, sin entrar en detalles en la física y la inteligencia artificial, tratados en otros anexos.

## A.1. Introducción

Todos los videojuegos, independientemente del tipo del mismo, suelen tener un diseño básico similar.

Desde el punto de vista de la programación, el principal componente de todo juego es el bucle principal del juego. Este bucle se ejecuta constantemente mientras el juego está en funcionamiento y se encarga de, teniendo en cuenta la posible interacción del usuario, recalcular el nuevo estado del juego y representarlo gráficamente.

El bucle principal en un juego suele tener un aspecto similar a este:

```
while (!end_of_game) {
    get_input();
    calculate_new_state();
    draw_output();
}
```

Además del bucle principal, prácticamente todo juego requiere un sistema de menús para configurar ciertos aspectos del mismo, así como algún tipo de sistema de pausa para poder parar la ejecución del juego temporalmente.

Otro componente importante en un videojuego es el sistema de control del mismo y la gestión de dispositivos de entrada, como el ratón o el teclado, necesaria para este propósito.

En este anexo trataremos el diseño general del juego, centrándonos en los tres aspectos recién mencionados y dejando otros más específicos de este juego en general, como la física o la inteligencia artificial, para posteriores anexos.

## A.2. Estructura general del juego

A.1.

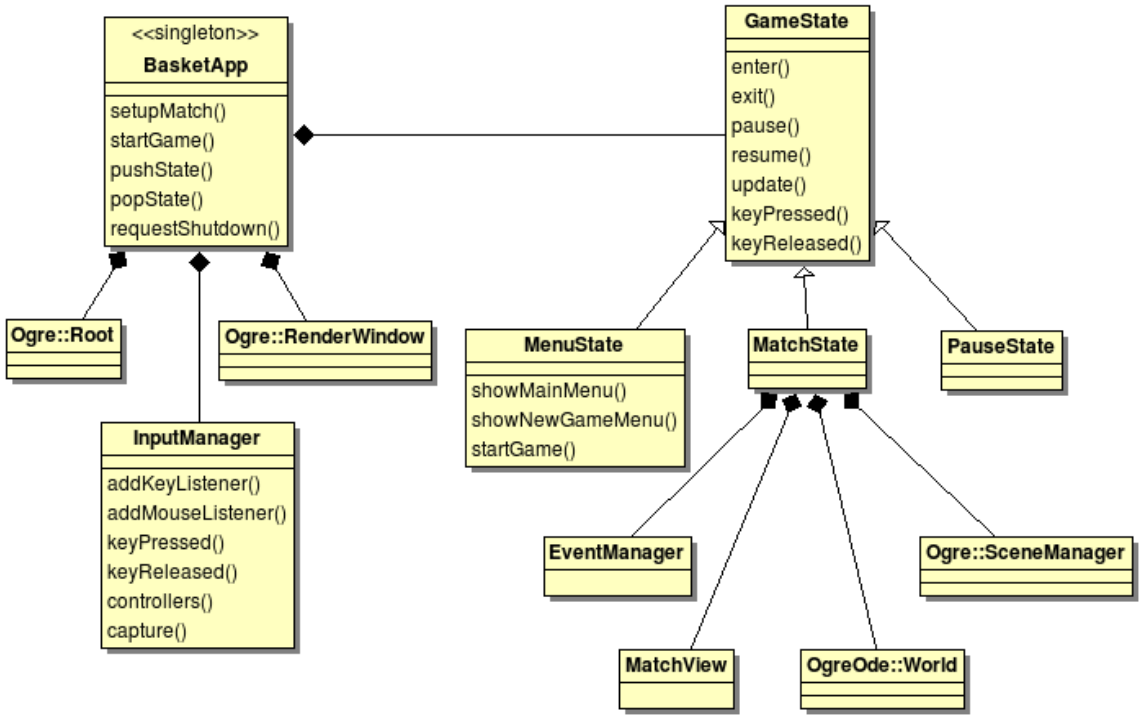



Figura A.1: Diagrama de clases principal

La figura A.1 muestra las clases más importantes en el diseño general del juego. En primer lugar podemos observar la clase *BasketApp*. Ésta es la clase principal del juego, usada desde la función *main()* para lanzar el juego, quitando la gestion de argumentos de línea de órdenes y la gestión de excepciones, el código es:

```
int main(int argc, char **argv) {
    BasketApp *app = BasketApp::getSingletonPtr();
    app->startGame();
}
```

El método *startGame* en primer lugar inicializa el sistema gráfico, después carga los estados fundamentales del juego (menú, partido y pausa), inicializa el gestor de datos (acceso a la base de datos de equipos) y el gestor de entrada de datos, añadiéndo a la propia clase como receptora de eventos de teclado y ratón.

Una vez realizada la inicialización, se establece el estado inicial del juego (menú o partido, dependiendo de las opciones de arranque). La gestión de estados se realiza mediante una pila, mediante los métodos *pushState()*, *popState()* y *changeState()*.

El bucle principal es el siguiente:

```
while(!mShutdown){
    unsigned long timeCurrentFrame =
       mRoot->getTimer()->getMilliseconds();
    unsigned long timeSinceLastFrame =
       timeCurrentFrame - timeLastFrame;
    timeLastFrame = timeCurrentFrame;

    if(mPushScheduled){
        this->pushState(mStateToPush);
        mPushScheduled = false;
        mStateToPush = 0;
    } else if(mPopScheduled) {
        this->popState();
        mPopScheduled = false;
        if(mStates.isEmpty()) {
            return;
        }
    }
    mInputManager->capture();
    mStates.top()->update(timeSinceLastFrame);
    mRoot->renderOneFrame();
    Ogre::WindowEventUtilities::messagePump();
}
```

En primer lugar, se comprueba si se ha realizado una petición de finalizar el juego mediante el método *requestShutdown()*. En caso contrario se continúa con la ejecución del bucle. El tiempo que pasa entre una ejecución del bucle y la siguiente es variable, y la actualización de los elementos del juego debe tener en cuenta este tiempo, por lo que se calcula al principio del bucle.

A continuación se realiza la gestión de estados, comprobando si durante la ejecución anterior del bucle algún elemento del juego solicitó insertar un nuevo estado o volver al estado anterior. En caso de que así sea, se llama al método apropiado. Al ejecutar *pushState()*, se pausa el estado actual llamando al método *pause()* del estado en el tope de la pila, se añade el nuevo estado y se inicia su ejecución mediante el método *enter()*. Si por el contrario se ha solicitado volver al estado anterior, el método a ejecutar será *popState()*. En este caso se ejecutará el método *exit()* del estado en el tope de la pila, se eliminará este estado de la misma y se ejecutará el método *resume()* para el estado que ahora esta en el tope de la pila.

Como podemos ver, los cambios de estado no se realizan directametente desde los estados, si no que se solicitan mediante los métodos *schedulePushState()* y *schedulePopState()*. La razón para ésto es que de hacerlo desde el propio estado, estaríamos parando la ejecución del antiguo estado a mitad de ejecución, lo cual puede ser problemático dependiendo del contexto en el que se encuentra el estado. Al usar un mecanismo de solicitud, nos aseguramos de que el cambio de estado se realiza cuando métodos del mismo no están ejecutándose.

A continuación se realiza la captura de dispositivos de entrada por parte del gestor de entrada, utilizando el método *capture()* de la clase *InputManager*. Una vez que hemos adquirido la información de los dispositivos de entrada, podemos pasar a ejecutar la actualización del estado actual del juego.

Cada uno de los diferentes estados implementa el método *update()* y el bucle principal se limita a llamar a este método para el estado que se encuentra en el tope de la pila, pasando como parámetro el tiempo transcurrido desde el último frame. Una vez que esta llamada a actualizado el estado del juego, ejecutamos el método *renderOneFrame()* para mostrar por pantalla el estado actual del juego.

El primer estado en mostrarse en una ejecución normal del juego es *MenuState*, encargado de mostrar los menús iniciales del juego. El siguiente estado en el orden habitual de ejecución es *MatchState*. Este estado es el más complejo del juego e instancia el gestor de eventos, el partido y la vista del partido y otras clases necesarias para visualizar esta parte del juego. Estas clases se examinan en más profundidad en el anexo C. Por último, el tercer estado y el más simple de ellos, es *PauseState*, que simplemente se utiliza para parar la ejecución del estado de partido.

Hemos comentado con anterioridad como la clase *BasketApp* se ha registrado para recibir los eventos de teclado y ratón. En realidad esta clase sólo actúa como retransmisora de los eventos, haciéndolos llegar al estado en el tope de la pila, que debe saber como reaccionar a ellos.

# Anexo B

# Física

En este anexo se ofrecen detalles sobre la resolución de los problemas relacionados con la simulación física en el juego y los detalles de su implementación utilizando ODE.

## B.1. Uso de ODE en In Your Face

El motor de física ODE se ha utilizado para simular el comportamiento físico del balón de baloncesto, incluyendo su rebote con los diferentes componentes de la cancha de baloncesto, como el suelo, los tableros y los aros. También se ha utilizado para determinar las colisiones de los jugadores, tanto entre ellos mismos como con el balón.

Los objetos están simulados utilizando sólidos con una masa y geometría asociadas. La masa es utilizada para simular el comportamiento físico del objeto, mientras que la geometría se utiliza para la detección de colisiones.

El balón está simulado utilizando un sólido con la masa repartida esféricamente y una geometría esférica.

```
OgreOde::SphereMass mass(7.0,ballRadius);
mBody = new OgreOde::Body(mWorld);
mBody->setMass(mass);
mNode->attachObject(mBody);

mGeom = (OgreOde::Geometry*)new OgreOde::SphereGeometry(
        ballRadius, mWorld, mWorld->getDefaultSpace());
mGeom->setBody(mBody);
mGeom->setUserAny(Ogre::Any((ObjectView*)this));
```

Los objetos *OgreOde::SphereMass* y *OgreOde::SphereGeometry* se utilizan para generar una masa y una geometría esféricas respectivamente.

El suelo de la pista es estático, por lo que solamente necesita una geometría plana para poder detectar los rebotes del balón e impedir que los jugadores caigan en el vacío.

```
mGeom = new OgreOde::InfinitePlaneGeometry(plane,
    mWorld, mWorld->getDefaultSpace());
```

```
mGeom->setUserAny(Ogre::Any((ObjectView*)this));
mEntity->setUserAny(Ogre::Any((OgreOde::Geometry*)mGeom));
```

Los jugadores podrían ser muy complejos, pero nuestra aproximación simple para un jugador consta de un cilindro para el cuerpo con dos cilindros móviles para los brazos.

Por defecto, las geometrías cilíndricas en ODE son cilindros cuyo eje es paralelo al eje Y. Para conseguir un cilindro en posición vertical es necesario rotarlo 90 grados respecto al eje X. Para no tener que tener esto en cuenta a lo largo de todo el código, creamos una geometría transformada y aplicamos la transformación que acabamos de mencionar. A partir de ese momento el cilindro siempre estará rotado 90 grados respecto al eje X dentro de la geometría transformada y podemos operar con esta geometría de manera normal, asumiendo que su posición normal es la de un cilindro vertical (eje paralelo a la dirección del eje Z).

```
OgreOde::CylinderMass mass(5.0, Ogre::Vector3::UNIT_Y, 0.25, 2);
OgreOde::CylinderMass armMass(1.0, Ogre::Vector3::UNIT_Y, 0.1, 0.8);

mBody = new OgreOde::Body(mWorld);
mBody->setMass(mass);
mBody->setAffectedByGravity(false);

mSpace = new OgreOde::SimpleSpace(mWorld,mWorld->getDefaultSpace());
mSpace->setInternalCollisions(false);

mTransGeom = new OgreOde::TransformGeometry(mWorld,mSpace);
mGeom = (OgreOde::Geometry*)
    new OgreOde::CylinderGeometry(0.25,1.5,mWorld);
mGeom->setOrientation(
    Ogre::Quaternion(Ogre::Radian(Ogre::Degree(90)),
    Ogre::Vector3::UNIT_X));
mTransGeom->setEncapsulatedGeometry(mGeom);
mTransGeom->setBody(mBody);
mTransGeom->setUserAny(Ogre::Any((ObjectView*)this));
mGeom->setUserAny(Ogre::Any((ObjectView*)this));

for(int i=0;i<2;i++) {
    mArmGeoms[i] = (OgreOde::Geometry*)
        new OgreOde::CylinderGeometry(0.1,0.8,mWorld,mSpace);
    mArmWires[i] = new Ogre::WireBoundingBox();
    mArmWires[i]->setupBoundingBox(mArmGeoms[i]->getAxisAlignedBox());
    mArmWireNodes[i] = mNode->createChildSceneNode();
    mArmWireNodes[i]->attachObject(mArmWires[i]);
    mArmNodes[i] = mNode->createChildSceneNode();
    mArmNodes[i]->setInheritOrientation(false);
    mArmNodes[i]->attachObject(mArmEntities[i]);
}
```

El aro es el objeto más complejo desde el punto de vista de la representación geométrica con ODE, ya que no puede utilizarse ninguna de las geometrías básicas. Además el realismo del rebote del balón con el aro es muy importante para la apariencia del juego.

La opción más sencilla es la creación de una geometría de malla de triangulos a partir de la malla utilizada para la visualización del aro. Esta primera aproximación ha resultado funcionar de manera adecuada por lo que no se ha investigado más.

```
OgreOde::EntityInformer ei(mEntity);
mGeom = ei.createStaticTriangleMesh(mWorld,mWorld->getDefaultSpace());
mGeom->setUserAny(Ogre::Any((ObjectView*)this));
```

Los usos más importantes de la deteccion de colisiones son la simulación del rebote en el aro, la intercepción de pases, la decisión sobre cual de los jugadores en disputa por un balón perdido ha llegado antes a él y colisiones entre jugadores que pueden ser infracciones del reglamento.

A continuación vamos a ver como usar las geometrías creadas con anterioridad para detectar las colisiones y actuar en consecuencia.

En algunas de las secciones de código anteriores, podemos ver líneas como:

```
mGeom->setUserAny(Ogre::Any((ObjectView*)this));
```

Estas líneas asocian las geometrías con el objeto al que pertenecen, de manera que cuando dos geometrías colisionan podemos saber a que objetos representan. El método *collision* recibe un parámetro de tipo *OgreOde::Contact*, a partir del cual podemos obtener las geometrías envueltas en la colisión:

```
bool MatchView::collision(OgreOde::Contact *contact) {
    OgreOde::Geometry *g1 = contact->getFirstGeometry();
    OgreOde::Geometry *g2 = contact->getSecondGeometry();
```

A partir de las geometrías podemos obtener los objetos asociados, utilizando el método *getUserObject*. Como todos los objetos del juego heredan de la clase base *ObjectView* podemos usar una conversión a esta clase en principio y después determinar el tipo exacto de objeto:

```
obj1 = Ogre::any_cast<ObjectView *>(g1->getUserAny());
if (obj1 == mBallView) {
```

Algunas de las acciones que se realizan ante colisiones son las siguientes:

- Colisión entre balón y pista

```
si la posesión ha expirado:
   lanzar evento de posesión expirada
sino:
   si la colisión es fuera de la pista:
      lanzar evento de fuera de banda
```

- Colisión entre balón y aro

```
iniciar una nueva posesión
```

- Colisión entre balón y jugador

```
si la posesión ha expirado:
   lanzar el evento de posesión expirada
sino:
   modificar el estado del juego para
   que el jugador posea el balón
```

- Colisión entre 2 jugadores

```
si uno de los dos está parado:
   si el otro se está moviendo por
         encima de un límite:
      falta
sino:
   si uno tiene la bola:
      si el otro se está moviendo por
            encima de un límite:
         falta
```

# Anexo C

# Inteligencia artificial

En este anexo se ofrecen detalles sobre el diseño de la inteligencia artificial para el juego.

## C.1. Introducción

La inteligencia artificial es la ciencia que se encarga de estudiar la creación de máquinas inteligentes y, en especial, programas inteligentes. Podemos decir que un programa o un agente de un programa es inteligente si tiene la capacidad de estudiar su entorno y reaccionar de la manera correcta para lograr algunos objetivos de una manera similar a la que lo haría un ser humano.

Cuando hablamos de diseñar una inteligencia artificial para un juego, estamos haciendo referencia al hecho de proveer al programa de los mecanismos suficientes para que pueda comportarse de una manera aparentemente inteligente.

A continuación estudiaremos la arquitectura básica de la inteligencia artificial desarrollada y entraremos en detalle sobre las distintas técnicas utilizadas, como máquinas de estados, IA jerárquica y *steering behaviours*. También expondremos algunos de los problemas geométricos y físicos básicos encontrados en la implementación y como se han resuelto.

## C.2. Arquitectura básica de la inteligencia artificial

La arquitectura básica de la IA está reflejada en la figura C.1.

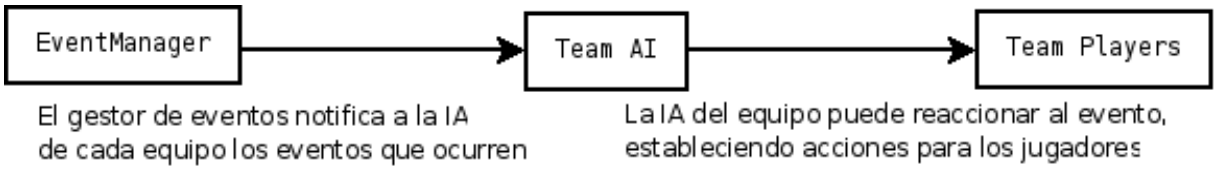



Figura C.1: Diagrama de alto nivel de la IA

El gestor de eventos es el encargado de distribuir los eventos que se generan en los distintos puntos de la aplicación. Estos eventos llegan a las inteligencias artificiales de los equipos, las cuales, después de consultar el estado del resto de elementos del juego, establecen las acciones a ejecutar por los jugadores.

## C.3. Gestor de eventos

La inteligencia artificial necesita obtener información de su entorno para poder calcular la mejor forma de actuar. Podemos intentar obtener esta información utilizando dos aproximaciones: encuesta y sistemas de eventos.

El método de encuesta se limita a que el código de la IA busque la información adecuada en el momento que la necesita. Este método es adecuado en algunas situaciones como, por ejemplo, la evaluación de las probabilidades de éxito al decidir si realizar un pase a un compañero de equipo.

Sin embargo, hay información para la que este método no es óptimo. Por ejemplo, no sería razonable que en cada ejecución de la IA debíeramos comprobar si tenemos la posesión del balón o no. En estos casos es más apropiado utilizar un sistema de eventos. En el ejemplo anterior, cuando un jugador nuevo se hace con el control de la bola, se lanza un evento *GotBall* que la IA recibe y es en ese único momento cuando la IA puede comprobar si ha habido un cambio de posesión o no. Mientras no llegue un evento de cambio de posesión, la IA puede seguir ejecutándose presumiendo que no ha habido tal cambio.

En el proyecto se ha utilizado la aproximación del sistema de eventos, aunque la IA utiliza encuesta para obtener más información para saber como reaccionar correctamente a un evento.

La arquitectura del sistema de eventos está reflejada en la figura C.2.

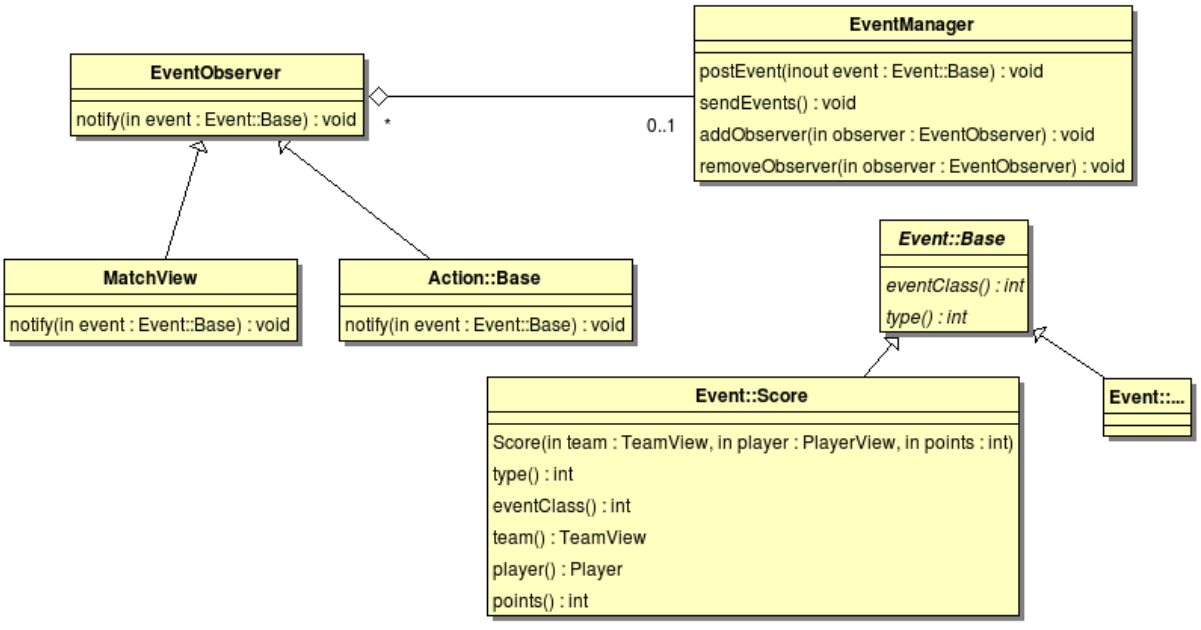



Figura C.2: Diagrama del sistema de eventos

Los eventos están definidos en su propio espacio de nombres, *Event* y representan todos los eventos del partido en los que la IA puede estar interesada. Cada evento puede incluir datos relevantes al mismo. A continuación se muestran algunos ejemplos de eventos:

- *MatchStart*: se genera de manera artificial para comenzar el partido
- *Second*: se genera cada segundo
- *GotBall*: se genera cuando un jugador recibe el balón, incluye el jugador que lo ha recibido
- *FreedBall*: se genera cuando un jugador suelta el balón, ya sea para pasarlo o realizar un tiro a canasta, e incluye el jugador que lo ha soltado
- *Input*: se genera cuando un humano pulsa o suelta una de las teclas asociadas a un controlador, incluye la función de la tecla y si ha sido pulsada o soltada
- *FailedShot*: se genera cuando un tiro a canasta falla, es decir, baja por debajo del nivel del aro sin entrar por el mismo
- *ActionFinished*: se genera cuando un jugador completa la acción que tenía asignada, incluye el jugador que ha realizado la acción

Como vemos, algunos eventos representan hechos reales del partido, mientras que otros son artificiales, como los de entrada de datos por parte de un jugador humano.

Estos eventos se generan desde distintas partes de la aplicación y se envían al gestor de eventos (*EventManager*) utilizando el método *postEvent*. Éste encola los eventos recibidos para procesarlos en el bucle principal de la aplicación.

Cualquier objeto del juego puede implementar la interfaz *EventObserver*, basada en el patrón de diseño *Observer*, para recibir notificaciones de eventos. Durante el procesado de eventos, el gestor envía por orden todos los eventos encolados a todos los observadores que se han registrado para ello.

En la práctica el principal observador de eventos es la clase principal del partido, *MatchView*. Esta clase se encarga de procesar los eventos oportunos y redistribuir todos los eventos a los dos equipos. Existía la opción de que los equipos fueran también observadores, en lugar de distribuir los eventos desde el objeto del partido, pero de este modo podemos estar seguros de que la clase que representa al partido recibe los eventos antes que los equipos, pudiendo reaccionar a ellos en primer lugar.

Cuando la clase equipo recibe un evento, ésta notifica a la IA de manera que pueda alterar las acciones de los jugadores basándose en este nuevo evento en caso de que sea necesario.

# C.4. IA jerárquica

A la hora de programar la inteligencia artificial del juego es necesario tener en cuenta desde las decisiones estratégicas de equipo (¿qué jugada utilizar?, ¿qué hacer si pierdo de 3 puntos y quedan 5 segundos?) hasta los detalles más obvios (no salirse del campo con el balón) pasando por decisiones personales de dificultad intermedia (¿con qué ángulo debo tirar a canasta?).

Como se puede apreciar, los tipos de decisiones que ha de tomar la IA son muy diferentes entre sí, por lo que lo más aconsejable es tener una IA jerárquica, con diferentes capas que se ocupen de las distintas clases de decisiones.

A continuación introduciremos cada una de estas capas, empezando por las de más bajo nivel.

## C.4.1. Acción básica y *steering behaviours*

La acción básica y los *steering behaviours* representan la parte más básica del movimiento de los jugadores.

Una acción básica es una estructura de datos capaz de representar una acción de bajo nivel que un jugador debe realizar. Estas acciones son:

- *Stand*: permanecer parado
- *BasicMove*: movimiento básico
- *BasicPass*: pase
- *Turn*: giro
- *Jump*: salto
- *BasicShoot*: tiro básico

Además la acción básica contiene información sobre la dirección del tiro o pase y la posición en la que el jugador debe colocar sus brazos.

Aunque en principio estas acciones básicas parecen suficiente para realizar los movimientos de los jugadores, tan pronto como se ejecuta una simulación nos encontramos con problemas obvios. Por ejemplo, tomemos el problema de jugadores chocando entre ellos, algo que ocurre con mucha frecuencia. Para solucionar esto con las acciones básicas, sería necesario comprobar manualmente de manera continua si estamos a punto de chocar con un jugador, y en ese caso modificar la acción de movimiento para evitar el choque, y después restaurar el destino original del jugador cuando ya no hay riesgo de colisión. Con esta solución creamos un nuevo problema, ya que es posible que al intentar evitar a un contrario, un jugador que controla el balón se salga del terreno de juego, por tanto sería necesario comprobar que la trayectoria establecida para evitar la colisión no sale del terreno de juego.

Resulta evidente que es necesario otra capa para tratar estos problemas de movimiento de una manera más comoda. Esta capa adicional son los *steering behaviours*.

Los *steering behaviours* (comportamientos de dirección) son diferentes modos de comportamiento que un agente puede tener en cuanto a la dirección y velocidad de su movimiento. Ejemplos de estos comportamientos pueden ser ir a una posición determinada, evitar obstaculos o perseguir a otro agente.

Combinando estos comportamientos simples se pueden obtener esquemas de movimiento complejos de una manera muy sencilla. De esta manera es posible asignar a los jugadores acciones estratégicas como seguir a un rival o ir a una posición determinada a la vez que mantiene el comportamiento básico, como evitar salirse del terreno de juego o minimizar los choques con otros jugadores.

Como solución al problema comentado anteriormente, todos los jugadores tendrían siempre asignados los comportamientos de evitar choques y no salirse del terreno de juego. Cuando queremos que un jugador vaya a una posición determinada se le asigna un nuevo comportamiento para ir a determinada posición. En cada paso del juego, el jugador consultara todos los comportamientos y obtendrá la dirección que debe seguir como la composición de todos ellos. Si no hay riesgo de colisión o de salirse del campo, la dirección vendrá determinada únicamente por el comportamiento de moverse a una posición dada. Si por el contrario hay riesgo de colisión, el comportamiento de evitar colisiones modificará la dirección para evitar el choque con los otros jugadores.

Existe una clase llamada *SteeringManager* que se encarga de gestionar varios comportamientos y generar el resultado final que determinará la dirección de un jugador. Esta es el sencillo método que realiza el cálculo:

```
Ogre::Vector3 SteeringManager::velocity() {
    Ogre::Vector3 vel = Ogre::Vector3::ZERO;
    Steering::Base *s;
    foreach(s, mSteerings) {
        vel += s->velocity();
    }
    return vel;
}
```

Algunos de los comportamientos implementados en el juego son:

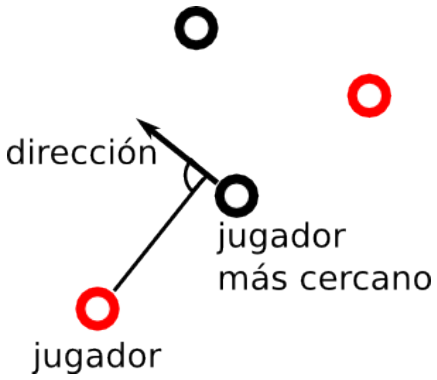
- *Seek* El comportamiento *seek* recibe como parámetro una posición destino que el jugador quiere alcanzar y en cada iteración devuelve la velocidad y dirección ideales para alcanzar ese punto. En la figura C.4.1 podemos ver la relación entre la dirección al destino y la velocidad normalizada.


Código:

```
Ogre::Vector3 Seek::velocity() {
    Ogre::Vector3 desiredVelocity;
    // if we are close enough to the desired point, stop
    if(player()->getPosition().positionEquals(mTarget,0.05)) {
        desiredVelocity = Ogre::Vector3::ZERO;
    // otherwise move towards it
    } else {
        desiredVelocity = mTarget - player()->getPosition();
        desiredVelocity[1] = 0;
        desiredVelocity.normalise();
    }
    return desiredVelocity;
}
```

- *Avoid* El comportamiento *avoid* no recibe parámetros y en cada iteración busca el jugador más próximo y en caso de estar en nuestra trayectoria devuelve una velocidad y dirección apropiadas para evitar la colisión.

  En la figura C.4.1 podemos observar como se realiza el cálculo de la dirección para evitar colisoines.



Código:

```
Ogre::Vector3 Avoid::velocity() {
    Ogre::Vector3 desiredVelocity = Ogre::Vector3::ZERO;
    Ogre::Vector3 pos = player()->getPosition();
    pos[1] = 0;
    MatchView *match = MatchState::getSingletonPtr()->matchView();

    // get closest player in a bounding box
    // in the direction the player is moving
    PlayerView *cp = match->getNearestCollidingPlayer(player());

    if(cp) {
        // get the matrix to change between absolute
        // coordinates and coordinates from the player's
        // point of view and its inverse
        Ogre::Matrix4 coord = player()->getCoordinateChangeMatrix();
        Ogre::Matrix4 inv = coord.inverse();
        Ogre::Vector3 np = coord * cp->getPosition();

        // if the opponent is less than a meter ahead of us
        if(np[2] < 1) {
            float dev = np[2];
            if(fabs(dev) < 0.1) {
                if(dev > 0) {
                    dev = 0.1;
                } else {
                    dev = -0.1;
                }
            }
            // calculate an avoiding direction inversely proportional
            // to the distance in player's view coordinates
            Ogre::Vector3 directiondest(0,0,-1/dev);

            // transform the direction to absolute coordinaates
            directiondest = inv * directiondest;
            Ogre::Vector3 direction = directiondest - pos;
            desiredVelocity += direction;
        }
    }
    return desiredVelocity;
}
```

## C.4.2. Acciones: Action::Base

Las acciones representan la siguiente capa en la inteligencia artificial.

Una acción representa una actividad de duración relativamente corta que un jugador puede realizar. A continuación se listan las acciones del juego:

- *BlockShot*: intentar realizar un tapón
- *Defend*: defender a un jugador, siguiéndolo por el campo
- *GetBall*: intentar coger el balón
- *InterceptBall*: intentar interceptar un pase
- *Move*: moverse en una determinada dirección
- *MoveTo*: moverse a una determinada posición
- *Pass*: pasar el balón a un compañero
- *Shoot*: intentar un tiro a canasta
- *Steal*: intentar robar el balón
- *Wait*: no hacer nada

Como se puede apreciar hay algunas acciones que ya se encontraban entre las acciones básicas, como la acción *Pass*. La diferencia entre ambas se encuentra en la inteligencia con la que cuenta cada una de ellas. La acción *Pass* sabe elegir la mejor manera de hacer un pase y puede no ser determinista, mientras que la acción básica *BasicPass* ya tiene determinados todos los parámetros del pase y se limita a realizarlo.

Todas las acciones del juego heredan de la clase *Action::Base*. Esta clase, además de determinar la interfaz que deben cumplir el resto de acciones, proporciona funcionalidad genérica a todas ellas, que describimos a continuación:

- Infraestructura básica

  Cada acción está asociada a un jugador, por lo que *Action::Base* proporciona métodos para establecer y obtener el jugador al que está asociado una determinada acción. Además cada acción es registrada en una tabla hash para poder obtener la acción a partir de su identificador. También se proporcionan las funciones para saber si una acción ha terminado y generar el evento apropiado.

- *Steering manager*

  El *steering manager* se encarga de almacenar los diferentes comportamientos que está acción emplea para describir su movimiento.

  Para saber la dirección final que el jugador debe tomar, se evalúa cada uno de los diferentes comportamientos, sumando cada una de las direcciones que devuelven para obtener una dirección única.

- Esperar un determinado tiempo

  ```
  void BaseAction::waitTime(double t)
  ```

  Es posible indicar que la acción debe terminar una vez pasado un número determinado de segundos.

- Esperar a que ocurra un evento global o un evento de jugador

```
void BaseAction::waitForGlobalEvent(Event::Type eventType);
void BaseAction::waitForPlayerEvent(Event::Type eventType,
                                    PlayerView *waitPlayer)
```

Es posible indicar que la acción debe terminar cuando se reciba un evento global o de jugador del tipo indicado. Para ello la acción se registra como observador en el gestor eventos.

- Esperar a que otra acción termine

```
void BaseAction::waitForAction(int actionId)
```

Es posible indicar que la acción debe terminar cuando otra acción termine. Cada acción tiene un contador de observadores, de tal modo que las acciones que tienen acciones dependiendo de ellas no son eliminadas hasta que el último de los observadores es consciente de que la acción ha terminado.

A continuación se muestra un ejemplo de uso:

```
//creamos una jugada de equipo
TeamPlay *t = new TeamPlay();

//creamos una lista de acciones para el jugador 0
ActionList *al = new ActionList();

//creamos una accion de movimiento para el jugador 0
Action::BaseAction *a = new MoveToAction(7,3);

//obtenemos el id para poder esperar despues
int moveActionId = a->id;

//la incluimos en la lista
al->append(a);
//incluimos en la jugada la lista de acciones
t->insert(0,al);

//creamos una lista de acciones para el jugador 1
ActionList *al = new ActionList();

//creamos una accion para esperar el movimiento del
//jugador 0
a = new WaitAction();
a->waitForAction(moveActionId);
al->append(a);

a = new MoveToAction(3,5);
al->append(a);
t->insert(1,al);
```

El jugador 0 ha acabado con una acción de movimiento, mientras que el jugador 1 tiene 2 acciones, la primera es esperar a que el jugador 0 acabe su movimiento, la segunda moverse a una posición determinada. Utilizando *waitForAction* podemos crear jugadas elaboradas donde los jugadores pueden esperar a que acciones de otros jugadores acaben antes de realizar algunas de sus acciones.

A continuación se muestran los métodos más importantes de cara a la inteligencia artificial que una acción debe proporcionar:

- `bool finished();`

  Determina si la acción ha terminado ya o no.

- `BasicAction run();`

  Transforma la acción en una acción básica y en *steering behaviours*.

  En muchas ocasiones el método *run* es complejo y está implementado mediante una máquina de estados, como en el caso de las acciones *Shoot* e *Intercept*. Para otras acciones, como *MoveTo*, este método es muy sencillo:

```
BasicAction MoveTo::run() {
    //we'll return a "default" BasicAction
    BasicAction a;
    //we just need to run this the first
    //time this method is called
    if(!mRun) {
        mRun = true;
        if(!(player()->hasBall() &&
           !MatchState::getSingletonPtr()
           ->matchView()->ballInPlay())) {
            //make sure we don't have any
            //previous steering behaviour
            steering()->clear();
            //add steering behaviour to
            //get to the desired position
            steering()->add(new Steering::Seek(mDest));
            //add steering behaviour to avoid obstacles
            steering()->add(new Steering::Avoid());
        }
    }
    return a;
}
```

- `Reaction processEvent(const Event::Base *event);`

  Permite que una acción reaccione a un evento, o bien de manera interna o, si es necesario un cambio externo, devolviendo una reacción. Una reacción es una estructura que permite cancelar la acción actual o reemplazarla con otra acción.

El control de jugadores por parte del jugador humano se realiza de esta manera, reaccionando a los eventos *Input*. Por ejemplo, en la acción *Shoot*, si llega un evento que indica que se ha soltado el botón de lanzamiento, el jugador suelta el balón, mientras que si llega un evento indicando que se ha pulsado el botón de pase, *processEvent* devuelve una reacción indicando que la acción actual debe reemplazarse por una acción de pase.

- `Ogre::Vector3 ballPosition();`

  Establece la posición de la bola respecto al jugador para esta acción en un determinado momento.

  Ejemplo para la acción *Wait*:

```
Ogre::Vector3 Wait::ballPosition() {
    Ogre::Vector3 pos = player()->node()->getPosition();
    return pos +
        Geom::bouncingBallPosition(player()->orientation());
}
```

  La función *Geom::bouncingBallPosition* simula el movimiento de botar el balón contra el suelo y es usado por varias de las acciones.

Cada jugador cuenta con una lista de estas acciones que realiza secuencialmente.Además los jugadores tienen una acción de espera que empieza a ejecutarse en caso de que su lista de acciones quede vacía.

A continuación estudiaremos la implementación de algunas acciones representativas.

#### *Action::Shoot*

La acción *Shoot* es la encargada de realizar los lanzamientos a canasta. Esta acción tiene varios estados que se muestran en en diagrama mostrado en la figura C.3.

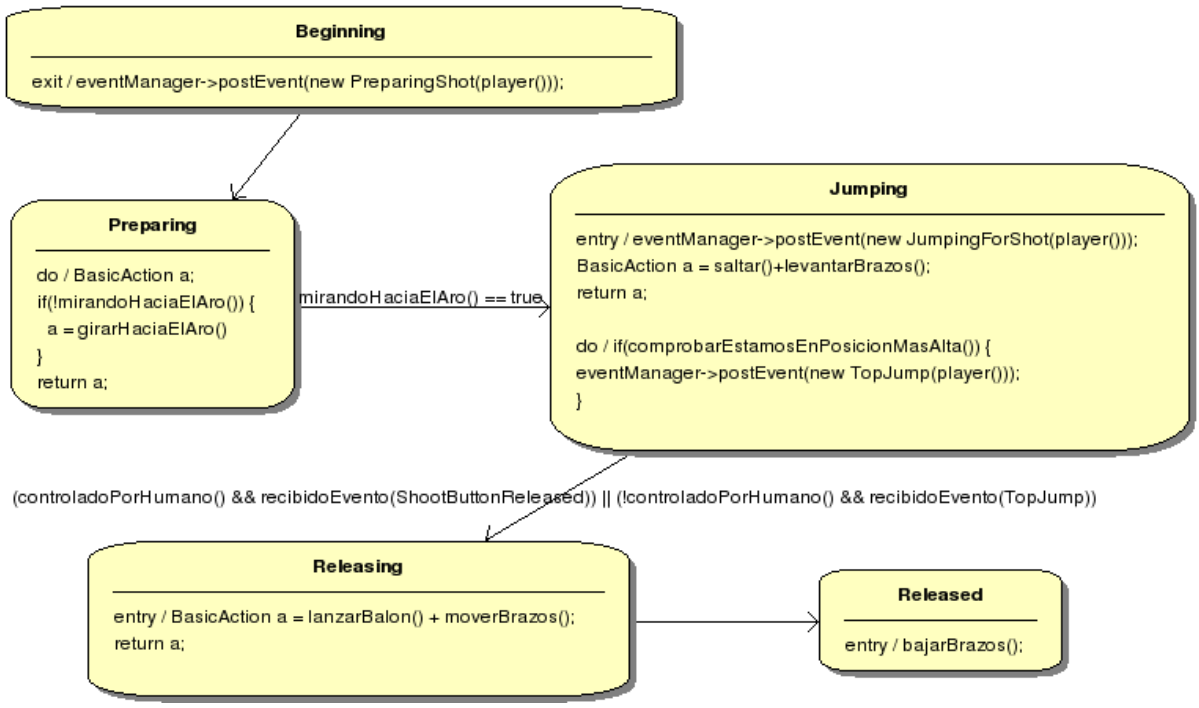



Figura C.3: Diagrama de estados de la acción *Shoot*

El estado *Preparing* se encarga de orientar al jugador hacia el aro de manera que pueda realizar el tiro, utilizando para ello una acción básica de tipo *Turn*. Una vez que está correctamente orientado se realiza la transición al estado *Jumping*. Al entrar a este estado se lanza el evento *JumpingForShot*, el propósito de este evento es que el adversario pueda reaccionar a él e intentar hacer un tapón.

Cuando el jugador llega al máximo punto de elevación en el salto se genera un evento *TopJump*. Este punto es el mejor para realizar el tiro y es en el que la IA lo realiza. Si el jugador que está lanzando es humano, la transición al estado *Releasing* se dispara cuando suelta el botón de lanzamiento.

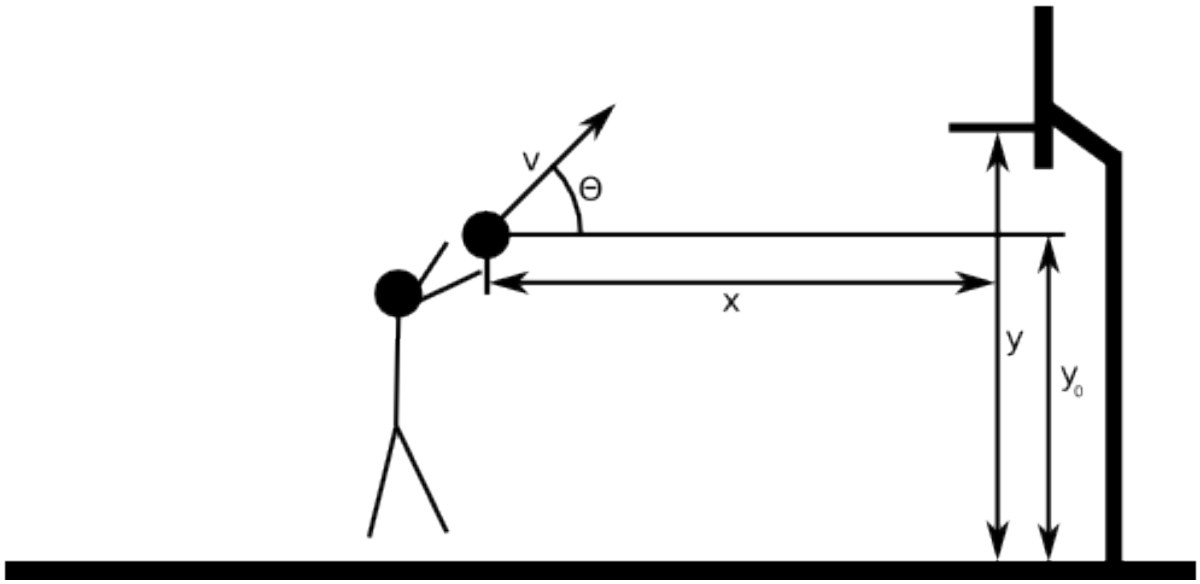



Figura C.4: Variables involucradas en el cálculo de la trayectoria de tiro

Considerando las variables que se muestran en la figura C.4, la trayectoria del tiro se calcula utilizando la siguiente fórmula: $y = y_0 + x \tan\theta - \frac{gx^2}{2(v\cos\theta)^2}$

Las incógnitas en esta ecuación son el ángulo ($\theta$) y la velocidad ($v$) de tiro. En primer lugar se determina el ángulo más adecuado para realizar el tiro en función de la distancia a la que nos encontramos de la canasta. Por el momento no se toma en cuenta el resto del entorno, pero puede resultar apropiado modificar el ángulo de tiro si hay oponentes cerca del tirador.

Una vez determinado el ángulo de tiro sólo es necesario obtener la velocidad a la que debemos lanzar el balón a partir de la ecuación. Este ángulo y velocidad nos da un lanzamiento óptimo que siempre se convertiría en canasta.

Por motivos obvios, este no es el comportamiento deseado, por lo que se realizan modificaciones sobre estos valores, teniendo en cuanto la habilidad de tiro del jugador, así como su velocidad en el momento de realizar el tiro. Por esto último, un tiro ideal se realiza parado y en el momento más alto del salto.

Para conferir un comportamiento no determinista al tiro, la desviación se multiplica por una distribución normal, de tal modo que la mayoría de los tiros tienen un pequeño desvío pero algunos de ellos puede desviarse más de lo normal.

### C.4.3. IAs de equipo: AI::Base

El nivel más alto de la IA es el que controla las acciones de todos los jugadores de un equipo. El juego está abierto a que se implementen más de una inteligencia artificial, por lo que hay una clase base llamada *AI::Base* que determina la interfaz que deben cumplir las inteligencias aritificiales que se escriban. La interfaz es muy simple y sólo consta de dos métodos:

- `virtual bool notify(const Event::Base &event) = 0;`

Se utiliza para notificar a la IA de un evento, en caso de que el evento necesite que se actualicen las acciones de los jugadores devolvera verdadero

- `virtual TeamPlay *getPlay() = 0;`

  Devuelve las acciones que los jugadores deben realizar

La IA puede consultar el estado completo del juego pero no realiza ninguna modificación directamente. Simplemente modifica su estado interno basándose en los eventos recibidos y proporciona la lista de acciones que los jugadores deben realizar en la estructura *TeamPlay*, que es capaz de almacenar una lista de acciones para cada jugador del equipo.

Cuando el equipo recibe un evento notifica a la IA y en caso de que ésta lo juzque necesario asigna las nuevas acciones a los jugadores:

```
void TeamView::notify(const Event::Base &event) {
    if(mAI->notify(event)) {
        assignPlay(mAI->getPlay());
    }
}
```

En este proyecto se ha implementado una inteligencia artificial de equipo que se ha denominado *Standard* y que se describe en la siguiente sección.

## C.5. Inteligencia artificial *AI::Standard*

La inteligencia artificial de equipo está basada en una máquina de estados.

Los estados deben heredar de la clase abstracta *TeamState::Base*, que define la interfaz que éstos deben cumplir, listada a continuación:

- ```
  virtual void enter() = 0;
  ```

  Método a ejecutar cuando se entra en este estado

- ```
  virtual void exit() = 0;
  ```

  Método a ejecutar cuando se sale de este estado

- ```
  virtual QString notify(const Event::Base &event) = 0;
  ```

  Notifica al estado sobre la llegada de un nuevo evento. La cadena devuelta indica el nuevo estado al que se debe cambiar. En caso de que no sea necesario un cambio de estado devolverá una cadena vacía.

- ```
  virtual TeamPlay *getPlay() = 0;
  ```

  Devuelve las acciones que deben realizar los jugadores según el estado actual del estado.

- ```
  virtual bool hasChanged() = 0;
  ```

  Devuelve verdadero si ha habido cambios en las acciones que los jugadores deben ejecutar desde la última vez que se consultaron.

Se pueden observar los estados básicos y sus transiciones por defecto en el diagrama C.5.
  ```

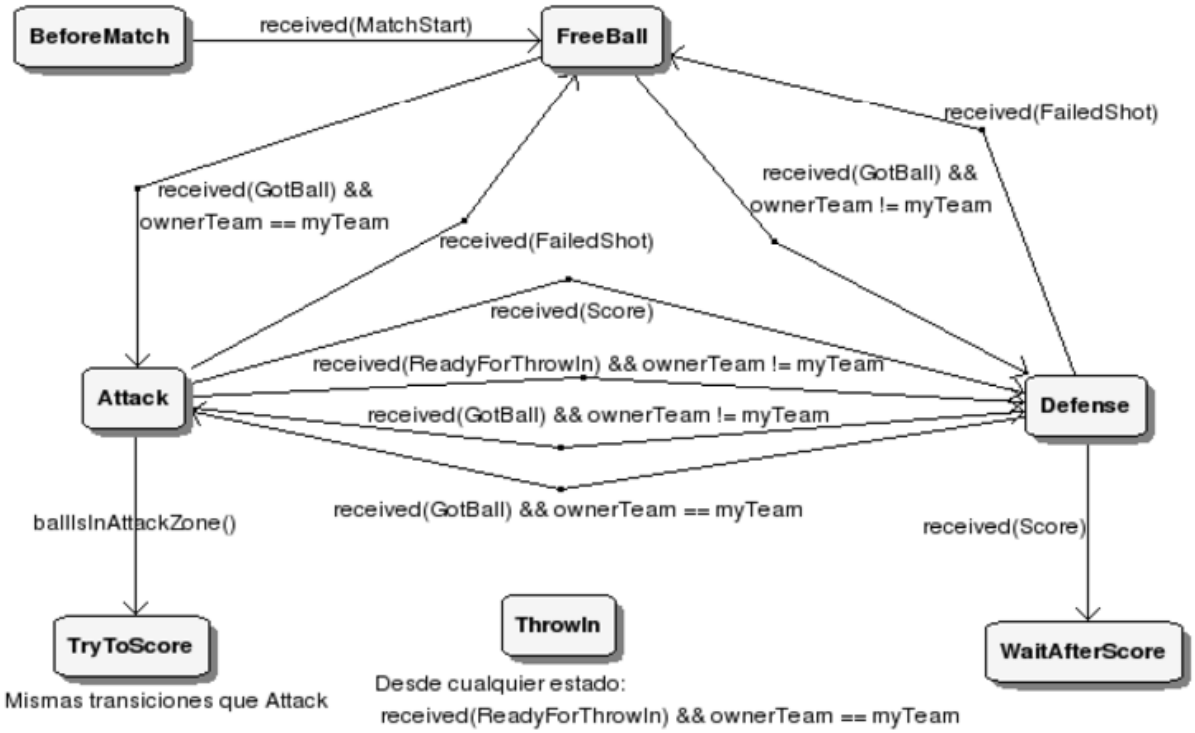



Figura C.5: Diagrama de estados básico de la IA

Cuando se lanza la IA se establece automáticamente el estado inicial, *BeforeMatch*. A continuación la IA se limita a recibir eventos y notificar al estado actual. Si el estado determina que es necesario un cambio, la IA ejecuta el método *exit()* del estado actual, cambia al estado siguiente y ejecuta el método *enter()* de éste.

El estado *BeforeMatch* se encarga de mover a los jugadores a las posiciones iniciales para el comienzo del partido. Cuando llega el evento *MatchStart* se pasa al estado *FreeBall*. Este estado es el que tienen ambos equipos cuando la bola no está en posesión de ninguno de ellos.

En este estado se examina la posición del balón en el campo y la distancia al mismo del jugador más cercano de cada equipo, para determinar cual es la probabilidad de que el equipo consiga la posesión. Basándose en esta información las reacciones pueden ser las indicadas en la tabla C.1.

| Probabilidad | Posición balón | Acción del equipo |
|---|---|---|
| p >0.8 | Cualquiera | El jugador intenta coger la bola y el resto del equipo se lanza al ataque |
| 0.8 >p >0.6 | Zona de ataque | El jugador intenta coger la bola y el resto del equipo se lanza al ataque |
| 0.8 >p >0.6 | Resto de zonas | El jugador intenta coger la bola y el resto del equipo espera |
| 0.4 >p >0.2 | Cualquiera | El jugador intenta coger la bola y el resto del equipo defiende |
| 0.2 >p | Cualquiera | El equipo completo defiende |

Tabla C.1: Resumen de acciones de *FreeBall*

Cuando un equipo consigue el balón pasa al estado *Attack*, mientras que el equipo contrario pasa al estado *Defense*.

El propósito del estado *Attack* es conseguir llevar el balón hasta la posición de ataque. En primer lugar, si el jugador que tiene el control de balón no es el base, se le pasa el balón a éste, de manera que sea él el encargado de llevar el ataque. En este estado se incluye la lógica necesaria para no incurrir en un *campo atrás*, esperando al base si es necesario.

Una vez se ha llegado a la posición de ataque se realiza una transición al estado *TryToScore*, cuyo objetivo es intentar conseguir una canasta.

En el estado *TryToScore* el jugador que posee el balón realiza una estimación de las posibilidades de encestar que tiene, evaluando diversos parámetros, como la distancia hasta la canasta, el hecho de que haya algún jugador que pueda taponar cerca y la habilidad del jugador. Además también se tiene en cuenta el tiempo de posesión que queda, de tal modo que, según se aproxima el final de la posesión, la probabilidad de que el jugador decida tirar crece.

Otro detalle a tener en cuenta durante el ataque es si nuestro lanzamiento va a ser de dos o tres puntos. Debido a la forma irregular del área de dos puntos no sería sencillo escribir una función para calcular si el jugador se encuentra en ella o no. La opción elegida para saber rápidamente el valor que tendría una canasta si un jugador lanza desde su posición actual, y también para poder realizar otras consultas similares necesarias por la inteligencia artificial como, por ejemplo, si un jugador está en la zona, ha sido usar un mapa de bits.

Para permitir crear diferentes pistas de manera sencilla se creó un script en Ruby usando la biblioteca ImageMagick que dibuja tanto la imagen de la pista normal como este mapa de bits. A continuación se muestra parte del código, en el que se dibujan las líneas que delimitan los campos, los círculos centrales y las partes del circulo del área pequeña:

```
#sidelines
gc.rectangle(0,0,$courtlength,$courtwidth)

#halfline
gc.line($horiz_center,0,$horiz_center,$courtwidth)

#center circles
gc.circle($horiz_center,$vert_center,
```

```
    $horiz_center+centersmallcircleradius,$vert_center)
gc.circle($horiz_center,$vert_center,
    $horiz_center+centercircleradius,$vert_center)

# ...

gc.push
# draws the continuous circle in the key
gc.clip_path('cont_key_circle_path')
gc.circle(579,$vert_center,579,579+365)
gc.pop
gc.push
gc.stroke('white')
gc.stroke_dasharray(41,36)
# draws the dashed circle in the key
gc.clip_path('dashed_key_circle_path')
gc.circle(579,$vert_center,579,579+365)
gc.pop
```

El mapa de bits se carga de manera sencilla en un objecto de tipo *Ogre::Image* durante la construcción del objeto que representa la pista:

```
mCourtBitmap.load("basketcourtmap.png",
    Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
```

Contamos con un método de utilidad para transformar entre las coordenadas de los jugadores en la pista y el pixel que representa esa posición en el bitmap:

```
Ogre::Vector3 CourtView::courtToBitmap(Ogre::Vector3 pos) {
    int width = mCourtBitmap.getWidth();
    int height = mCourtBitmap.getHeight();
    Ogre::Vector3 bitPos;
    bitPos[0] = pos[0]*100 + width/2;
    bitPos[2] = pos[2]*100 + height/2;
    return bitPos;
}
```

Para comprobar si la posición de un jugador está en la zona de tres puntos tenemos el siguiente método:

```
bool CourtView::posIsThreePointer(Ogre::Vector3 pos, CourtSide side) {
    if((side < 0) && (pos[0] < 0)) { return true; }
    if((side > 0) && (pos[0] > 0)) { return true; }
    Ogre::Vector3 bitPos = courtToBitmap(pos);
    Ogre::ColourValue c = mCourtBitmap.getColourAt((int)bitPos[0],(int)bitPos[
    if((c.r < 0.01) && (c.g < 0.01) && (c.b < 0.01)) {
        return true;
    }
    return false;
}
```

Como vemos, el método comprueba en primer lugar si el jugador está en la mitad de la pista opuesta a donde debe encestar, en cuyo caso siempre será un tiro de tres [2]. En caso contrario, se transforma su posición en la pista al pixel adecuado en el mapa de bits y se obtiene el color en ese punto. A continuación comprobamos si el color de la zona es negro (color asignado a la zona de tres puntos como se puede ver en el mapa de bits mostrado en la figura C.6) y devolvemos el valor adecuado dependiendo del color.

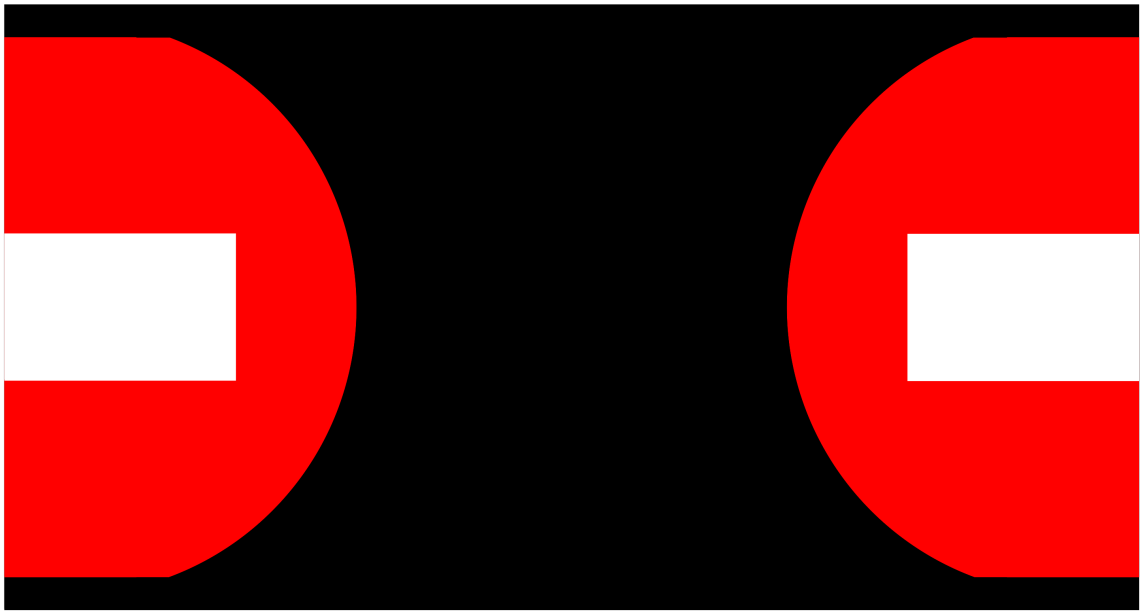


Figura C.6: Mapa de bits para diferenciar zonas de la pista

En caso de que no se considere apropiado realizar un tiro a canasta, el jugador que controla el balón estudiara la posibilidad de realizar un pase a un jugador que se encuentre en una mejor posición para tirar a canasta. El resto de jugadores tiene diversas posiciones a las que se van desplazando de manera aleatoria con el objetivo de facilitar un pase sencillo al poseedor del balón.

En el estado *Defense* cada jugador realiza un marcaje personal al jugador correspondiente del equipo atacante, utilizando la acción *DefendPlayer* para ello. Cuando el equipo contrario realiza un pase, se estudia la posibilidad de interceptarlo y, en caso de que parezca posible, el jugador con más posibilidades intenta alcanzar la bola.

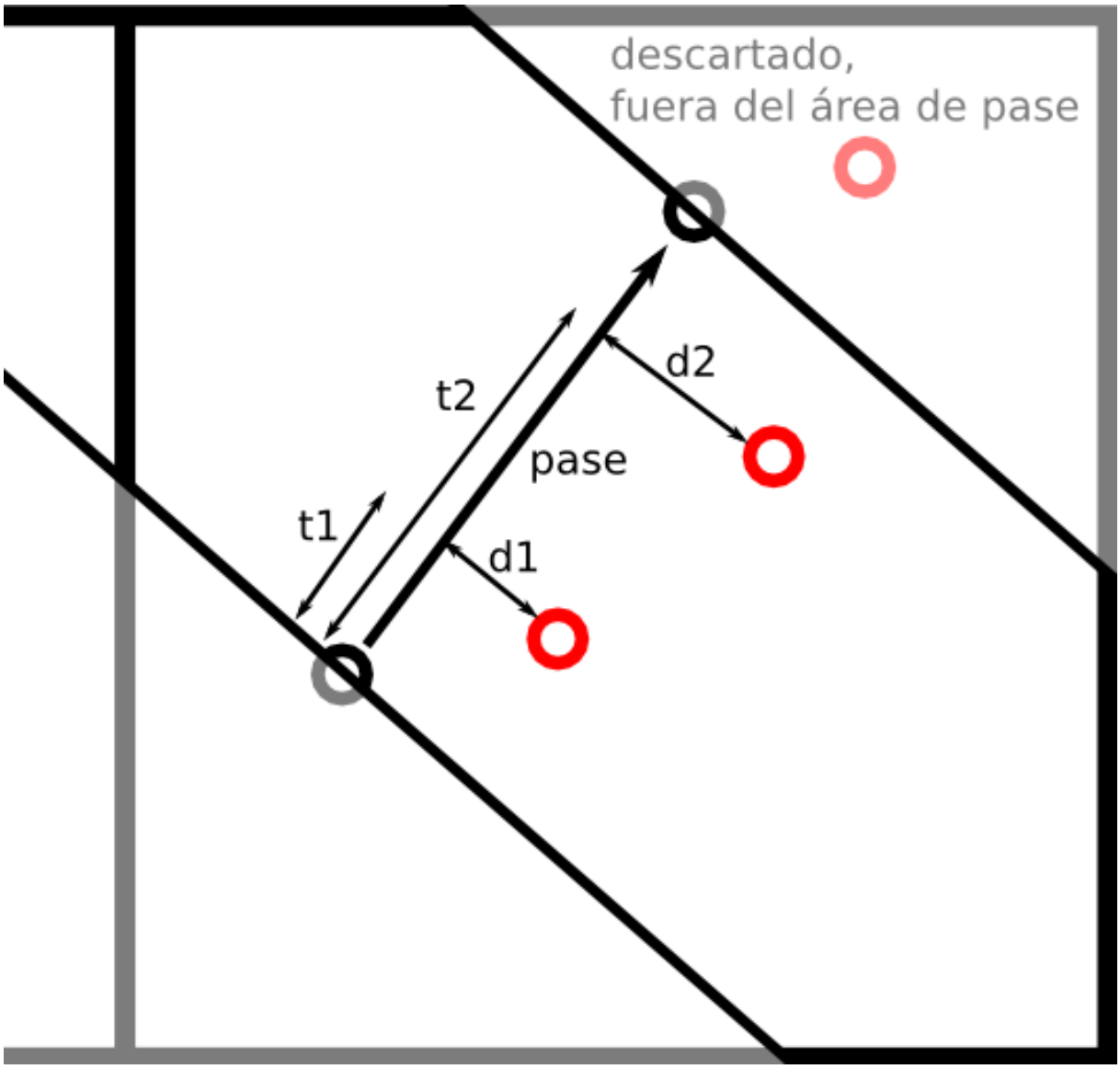



Figura C.7: Cálculo de la posibilidad de intercepción de pase

Para la intercepción del pase, como se puede apreciar en la figura C.7, en primer lugar se descartan los jugadores que no se encuentran entre los jugadores involucrados en el pase. Para los jugadores restantes, se calcula la distancia desde su posición hasta el punto más cercano a ellos en la trayectoria de la bola. Además se calcula el tiempo que a la bola le costará llegar desde las manos del jugador a esos posibles puntos de intercepción.

A continuación se incluye la función que realiza los cálculos geométricos para obtener toda la información necesaría sobre la intercepción:

```
Geom::InterceptionInfo Geom::passInterceptionInfo(
    Ogre::Vector3 ballPos, // initial position of the ball
    Ogre::Vector3 ballDir, // direction of the ball
    PlayerView *receiver,  // position of the player
                           // who will receive the ball
```

```
    TeamView *interTeam     // team that will try to
                            // intercept the ball
        ) {

    InterceptionInfo result;

    // we are not interested in the vertical speed of the ball for
    // interception purposes
    ballDir[1] = 0;

    Ogre::Vector3 recPos = receiver->node()->getPosition();

    // gather all information about the position of the
    // defensive players wrt the ball and the receiving player

    // discard the players which are not in the zone
    // between the ball and the receiving player

    // ballDir is the normal of both of the limiting planes
    // we also have a point for the first plane (ballPos) and
    // another one for the second one (recPos)

    // a plane is defined as A,B,C,D (A,B,C being the normal vector
    // and D defining the position along the normal)
    // the distance from a point to a plane is x*A+y*B+zC-D, where
    // x,y,z are the coordinates of the point

    // normalise ballDir to get a normal vector for both planes
    Ogre::Vector3 nBallDir = ballDir;
    nBallDir.normalise();
    result.ballDir = nBallDir;

    // get D for the ball plane and for the receiver plane
    double bpd = nBallDir.dotProduct(ballPos);
    double rpd = nBallDir.dotProduct(recPos);

    // get the normal of the plane in which the ball is moving and D
    // for that plane
    Ogre::Vector3 normal =
        Ogre::Quaternion(Ogre::Radian(Ogre::Degree(90)),
            Ogre::Vector3::UNIT_Y) * nBallDir;

    // as the plane we're interested in is the one who has ballPos on
    // it we know that x*A+y*B+zC-D equals 0, and thus we can get D
    normal.normalise();
    double bdd = normal.dotProduct(ballPos);

    // now we have all the information regarding the planes and can
    // check which players are actually between them
```

```
    for(int i=0; i<5;i++) {
        PlayerView *p = interTeam->playerView(i);
        double dotProduct =
            nBallDir.dotProduct(p->node()->getPosition());

        // distance to the ball plane
        double ballPlaneDistance = dotProduct-bpd;

        // distance to the receiver plane
        double receiverPlaneDistance = dotProduct-rpd;

        if((ballPlaneDistance > 0) && (receiverPlaneDistance < 0)) {
            result.players[i].betweenPlanes = true;
            receiverPlaneDistance = -receiverPlaneDistance;

            // now we know this player is between the planes and we
            // can calculate the distance to the plane of the ball
            // movement
            double ballDirectionDistance =
                normal.dotProduct(p->node()->getPosition())-bdd;

            // calculate the time it will take the ball to reach the
            // position in which the player should intercept it
            double timeLeft = ballPlaneDistance/ballDir.length();
            double distanceLeft = fabs(ballDirectionDistance) -
                timeLeft * playerSpeed;

            // a distance below 0 means that the player can actually
            // reach the intercepting position. note that an
            // interception is still possible even if distanceLeft is
            // greater than 0, as the player will use the arms to try
            // to reach the ball
            if(distanceLeft < 0) {
                distanceLeft = 0;
            }
            result.players[i].ballPlaneDistance = ballPlaneDistance;
            result.players[i].distanceLeft = distanceLeft;
            result.players[i].timeLeft = timeLeft;
        } else {
            result.players[i].betweenPlanes = false;
        }
    }
    // return the results from all the calculations to let the AI
    // decide if an interception should be attemped and which player
    // should try to perform it
    return result;
}
```

Una vez que tenemos todos estos datos es fácil calcular cual de los jugadores tiene más posi-

bilidades de alcanzar el punto de intersección antes de que la bola pase por él. En el caso del ejemplo, aunque el primer jugador se encuentra más cerca de la trayectoria de la bola, el segundo tiene más del doble de tiempo para recorrer la distancia que le separa de ella, por lo que tiene más probabilidades de interceptar el pase.

Además de estar pendiente de los intentos de pase, en el estado de defensa la IA también intenta realizar tapones ante los intentos de tiro. Mediante un algoritmo sencillo, donde la variable más influyente es la distancia al jugador que realiza el tiro, se decide cual de los jugadores es el más idoneo para intentar realizar el tapón.

El estado *WaitAfterScore* sólo se utiliza para esperar unos segundos tras una canasta antes de proseguir el juego.

El estado *ThrowIn* es el que guía las acciones de un equipo cuando tiene que volver a poner el balón en juego tras una canasta, una falta o un fuera de banda.

# C.6. Scripting de la inteligencia artificial *AI::Standard*

Una de las claves para la creación de una comunidad de contribuidores activa alrededor de un juego libre es que la barrera de dificultad para contribuir sea mínima.

El proceso de descargar todas las bibliotecas, realizar cambios en el juego y compilarlo no es trivial, por lo que, para proporcionar una manera sencilla de modificar el comportamiento de la inteligencia artificial, se ha creado una interfaz de scripting. Mediante esta interfaz se puede cambiar el comportamiento de los estados de la IA, así como añadir nuevos estados.

## C.6.1. Elección de un lenguaje y plataforma de scripting

Antes de implementar la interfaz de scripting se estudiaron los siguientes lenguajes de scripting con sus mecanismos asociados para exponer los objetos relacionados con la IA:

- *Boost.Python*. Python es un lenguaje de programación de propósito general. Existe una biblioteca del proyecto *Boost* llamada *Boost.Python* que permite interoperar fácilmente entre Python y C++.
- *LuaBind*. Lua es un lenguaje de programación diseñado específicamente para extender otras aplicaciones con una interfaz de scripting. LuaBind es una biblioteca similar a Boost.Python, y de hecho inspirada en ella, que permite integrar fácilmente Lua con C++.
- *QtScript*. QtScript es un lenguaje de scripting basado en el estándar ECMAScript (como, por ejemplo, JavaScript) incluido en la biblioteca Qt.

La principal ventaja de Boost.Python sobre el resto es que el lenguaje de programación subyacente es mucho más potente, contando con un gran número de bibliotecas de utilidades. Sin embargo, la ejecución de la IA debe ser muy rápida, por lo que los estados implementados no deberían ser demasiado sofisticados, y está ventaja no resulta por tanto demasiado importante.

Tras evaluar las tres plataformas, QtScript fue la elegida, en primer lugar por la facilidad de integración con el código existente, así como por el hecho de que el lenguaje QtScript, muy similar a JavaScript, es mucho más sencillo que los otros dos y, debido a su popularidad en la red, conocido por una mayor cantidad de posibles contribuidores.

## C.6.2. QtScript

QtScript es un motor de scripting basado en el estándar ECMAScript incluido en la biblioteca Qt desde la versión 4.

A continuación se muestra un ejemplo básico de ejecución de código ECMAScript con QtScript:

```
QScriptEngine engine;
qDebug() << "1 + 2 = " << engine.evaluate("1 + 2").toNumber();
```

En el ejemplo se pide al motor que ejecute el código ECMAScript "1 + 2$\tau$ devuelva el resultado.

Exportar una variable de manera que esté disponible en el entorno de ejecución QtScript también es muy sencillo:

```
QScriptValue val(&engine, 5);
engine.globalObject().setProperty("foo", val);
qDebug() << "foo * 2 = " << engine.evaluate("foo * 2").toNumber();
```

En este caso asociamos la variable C++ *val* con la variable QtScript *foo* y operamos con ella dentro del motor.

Del mismo modo podemos hacer disponible a QtScript clases de nuestro sistema, si bien debemos hacer que hereden de *QObject*, clase base de la mayoría de objetos de Qt.

La cuestión se complica ligeramente cuando entra en juego la herencia, como en las jerarquías de clases de acciones y eventos. En la siguiente sección veremos como se ha utilizado QtScript en el proyecto.

### C.6.3. Uso de QtScript en InYourFace

En primer lugar era necesario exportar a QtScript todas las clases que pueden ser interesantes para escribir un estado de la IA.

Las clases exportadas han sido:

- *Action* y toda la jerarquía de acciones
- *Event* y toda la jerarquía de eventos
- *MatchView*, clase que tiene acceso a ambos equipos, marcador, pelota, ...
- *TeamView*, clase que tiene acceso a los jugadores de un equipo
- *PlayerView*, clase que tiene acceso a toda la información de los jugadores
- *BallView*, clase que tiene acceso a la información de la pelota
- *TeamPlay*, clase que se utiliza para asignar una jugada determinada, construida a base de listas de acciones, a un equipo

Veremos, como ejemplo, como algunas de estas clases se han exportado. En primer lugar una de las clases más sencillas, *TeamPlay*.

El objetivo es poder ejecutar el siguiente código, en cuanto a la clase TeamPlay se refiere, en QtScript:

```
var i = 0;
var teamPlay = new TeamPlay();
var al = new ActionList();
teamPlay.insert(i,al);
```

Para ello se ha añadido una clase prototipo, que expone sólo las funciones deseadas y que además es necesaria para las clases que utilizan herencia. A continuación podemos ver el código añadido en los ficheros de la clase *TeamPlay*:

```
teamplay.h:
Q_DECLARE_METATYPE(TeamPlay *)

QScriptValue constructTeamPlay(QScriptContext *context,
    QScriptEngine *engine);

class TeamPlayPrototype :
        public QObject, public QScriptable {
    Q_OBJECT
    public slots:
        bool contains(const int key) const {
            TeamPlay *teamPlay =
                qscriptvalue_cast<TeamPlay *>(thisObject());
            return teamPlay->contains(key);
        }

        void insert(const int key, ActionList *value) {
            TeamPlay *teamPlay =
                qscriptvalue_cast<TeamPlay *>(thisObject());
            teamPlay->insert(key,value);
        }
};

teamplay.cc:
QScriptValue constructTeamPlay(QScriptContext *context,
        QScriptEngine *engine) {
    TeamPlay *teamPlay = new TeamPlay();
    return engine->toScriptValue(teamPlay);
}
```

La primera macro, *Q_DECLARE_METATYPE*, registra un tipo en el sistema de metatipos de Qt. Esto es necesario para poder transformar un puntero a un objeto TeamPlay a un objeto del lenguaje de scripting utilizando las funciones *toScriptValue*, *fromScriptValue* y el cast *qscriptvalue_cast*.

A continuación se declara la función *constructTeamPlay*, necesaria para poder crear objetos de tipo TeamPlay dentro del motor. Más tarde veremos como se asocia esta función en el motor para conseguir el objetivo deseado. Podemos ver que la implementación de la misma es muy sencilla.

Por último, declaramos una clase que hereda de QObject y de QScriptable e implementa dos métodos sencillos para consultar si un determinado jugador tiene una lista de acciones y para insertar una lista de acciones en la jugada.

Esto es todo lo necesario para hacer posible utilizar la clase TeamPlay desde nuestros scripts. A continuación veremos como exportar la jerarquía de eventos a QtScript, de tal manera que el siguiente código funcione correctamente:

```
function notify(ev) {
    if(ev.type() == BaseEvent.ActionFinished) {
        var af = new ActionFinishedEvent(ev);
        print(af.ActionId());
    }
}
```

Para ello, en primer lugar, en *event/base.h* añadimos un prototipo para la clase base incluyendo un método que nos permite obtener el tipo de evento:

```
class BasePrototype : public QObject, public QScriptable {
    Q_OBJECT
    public slots:
        int type() {
            const Base *event =
                qscriptvalue_cast<const Base *>(thisObject());
            return event->type();
        }
};
```

Añadimos un método que permite construir un objeto de tipo *ActionFinished* a partir de un evento base en QtScript:

```
QScriptValue Event::constructActionFinishedEvent(
        QScriptContext *context, QScriptEngine *engine) {
    QScriptValue value = context->argument(0);
    const Event::Base *ev =
        qscriptvalue_cast<const Event::Base *>(value);
    assert(ev->type() == ActionFinishedID);
    const Event::ActionFinished *afev =
        dynamic_cast<const Event::ActionFinished *>(ev);
    return engine->toScriptValue(afev);
}
```

Por último, creamos un prototipo alrededor de *Event::ActionFinished* para poder acceder al método *actionId*:

```
class ActionFinishedPrototype :
        public QObject, public QScriptable {
    Q_OBJECT
    public slots:
    QScriptValue actionId() {
        const ActionFinished *ev =
            qscriptvalue_cast<const ActionFinished *>(thisObject());
        return engine()->toScriptValue(ev->actionId());
    }
};
```

Para completar el soporte para scripting se ha creado un estado especial de la IA llamado *Scriptable*. El constructor de este estado acepta como parametro un nombre de un fichero que leera para obtener el script con la implementación del estado en QtScript.

Este estado cuenta con objeto QScriptEngine a través del cual se exporta el propio estado a QtScript, posibilitando el acceso al objeto que contiene toda la información del partido:

```
QScriptValue thisObject = mScriptEngine.newQObject(this);
mScriptEngine.globalObject().setProperty("state", thisObject);
```

Además se asocian los constructores a los tipos de datos para que podamos crear nuevos objetos desde el script:

```
TeamPlayPrototype *teamPlayPrototypeObject = new TeamPlayPrototype();
QScriptValue teamPlayProto =
    mScriptEngine.newQObject(teamPlayPrototypeObject);

mScriptEngine.setDefaultPrototype(
    qMetaTypeId<TeamPlay*>(),teamPlayProto);
QScriptValue teamPlayCtor =
    mScriptEngine.newFunction(constructTeamPlay, teamPlayProto);
```

Como último detalle para que todo funcione, cuando se crea un nuevo objeto de tipo *AI::Standard* se le puede pasar como parámetro el nombre de un directorio que contenga ficheros QtScript. En este caso para cada fichero que se detecte en ese directorio se creará un nuevo estado, a no ser que el estado ya exista, en cuyo caso se sustituye.

De esta manera podemos modificar fácilmente cualquier estado o añadir nuevos. A continuación se muestra un script que sustituye al estado attack, y que, una vez que los jugadores han llegado al lado de ataque, elige aleatoriamente entre otros scripts que contienen estados con jugadas concretas:

```
//standard_attack_notify con la respuesta estandar a eventos que
//suceden en el ataque, como Score, GotBall o FailedShot
function standard_attack_notify(ev) {
    var next = "";
    if(ev.type() == BaseEvent.GotBall) {
        var gb = new GotBallEvent(ev);
        if(state.team().name() != gb.owner().team().name()) {
            next = "Defense";
        }
    } else if (ev.type() == BaseEvent.Score) {
        next = "Defense";
    } else if (ev.type() == BaseEvent.FailedShot) {
        next = "FreeBall";
    } else if (ev.type() == BaseEvent.ReadyForThrowIn) {
        var rfti = new ReadyForThrowInEvent(ev);
        if(state.team().name() == rfti.team().name()) {
            next = "ThrowIn";
        } else {
            next = "Defense";
        }
    }
    return next;
}
```

```
attack.js:
// includes: standard_attack

var actions;
var finished;

function enter() {
    actions = new Array();
    finished = 0;

    //coordenadas de las posiciones a las que
    // queremos que vayan los jugadores
    var x = new Array(4,7,7,10,10);
    var y = new Array(0,4,-4,3,-3);

    var teamPlay = new TeamPlay();

    //poseedor de la bola
    var ballOwner = state.match().ball().owner().name();

    for(var i=0; i<5; i++) {
        var al = new ActionList();
        //si el jugador i tiene la pelota
        if (state.team().player(i).name() == ballOwner) {
            //y el jugador i no es el base
            if(i != 0) {
                //insertamos una accion para
                //pasarle la pelota al base
                var pg = state.team().player(0);
                var pa = new PassAction(pg);
                al.append(pa.toAction());
            }
        }
        //ademas incluimos una accion para moverse al sitio
        //que a cada jugador le corresponde
        var a = new MoveToAction(x[i],y[i]);
        var ba = a.toAction();
        al.append(ba);
        //almacenamos los identificadores de
        //las 5 acciones de movimiento
        actions[i] = ba.id();
        print("Action " + i + ": " + actions[i]);
        teamPlay.insert(i,al);
    }

    //establecemos la jugada y notificamos que ha cambiado
    state.setTeamPlay(teamPlay);
    state.setChanged();
```

```
}

function notify(ev) {
    //en primer lugar comprobamos si es uno de los eventos
    //que la funcion standard_attack_notify gestiona
    var next = standard_attack_notify(ev);
    if(next != "") {
        return next;
    }
    //en caso contrario, cada vez que acaba una accion
    if(ev.type() == BaseEvent.ActionFinished) {
        var af = new ActionFinishedEvent(ev);
        for (var i=0; i<5; i++) {
            //si era una de las acciones registradas anteriormente
            //incrementamos el contador
            if(af.actionId() == actions[i]) {
                finished++;
            }
        }
        //cuando todos han acabado de moverse

        if(finished == 5) {
            print("Ready to roll");
            //elegimos una de las 10 jugadas aleatoriamente
            random = Math.floor(Math.random()*10);
            //establecemos ese estado como nuevo estado
            next = "attack_play_" + random;
        }
    }
    return next;
}
```

## C.7. Métodos de depuración y mejora de la IA

Una vez tenemos una versión inicial de la inteligencia artificial, es importante ser capaz de depurarla cuando aparecen problemas, así como de introducir mejoras en ella. Para esto se utilizan tres métodos distintos.

En primer lugar, utilizando log4cpp, se ha creado un sistema de registro de información desde la aplicación. Este método está fundamentalmente orientado a la depuración de problemas. La información producida puede ser muy verbosa por lo que el sistema se ha organizado de manera jerárquica, haciendo sencillo activar y desactivar el registro de subsistemas del juego a distintos niveles. Esto permite obtener información de ejecución con gran cantidad de detalle de la parte concreta que queramos depurar.

Este es un ejemplo de uso del sistema de registros en el gestor de eventos, en el que sólo se registrará el evento en caso de que la categoría *eventmanager* esté activada:

```
log4cpp::CategoryStream debug =
    log4cpp::Category::getInstance("eventmanager").debugStream();

debug << "Posting event " << event->printable();
```

En segundo lugar, existe un sistema para mostrar en tiempo real información sobre la IA para cada jugador. Este sistema muestra etiquetas sobre los jugadores en el juego, incluyendo diferente información sobre el estado de la inteligencia artificial.

Una de las opciones es mostrar la acción que el jugador está realizando en ese momento, opcionalmente incluyendo los parámetros de la misma. Por ejemplo, si un jugador está moviéndose a una determinada posición, la información mostrada será:

```
Player 3
Moving to (3,8)
```

Esta opción se muestra en la figura C.8.

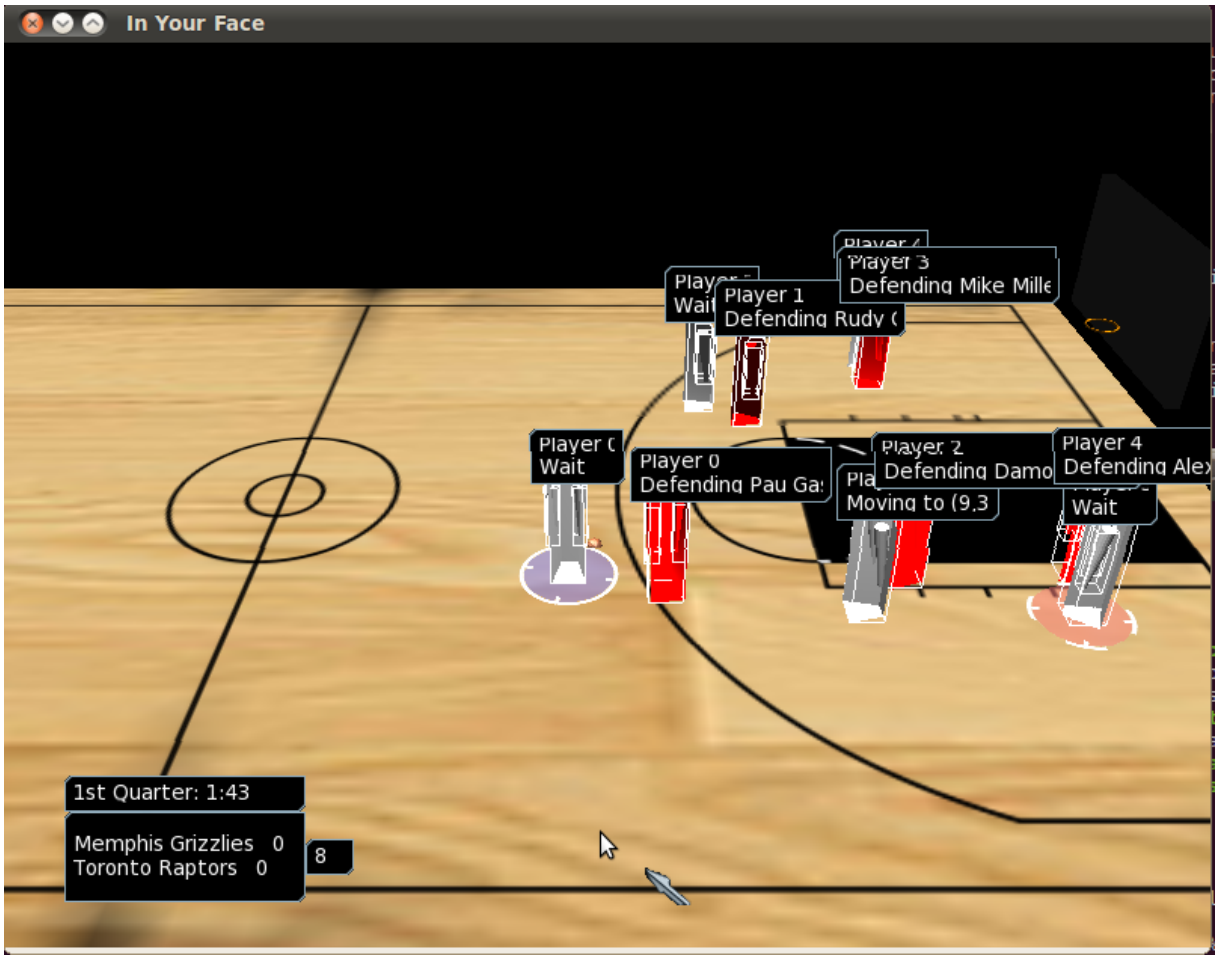



Figura C.8: Etiquetas de depuración de la IA

Las opciones adicionales pueden mostrar si la acción está terminada y si hay otras acciones que dependen de está acción.

Otra información muy interesante que podemos obtener con este método son las probabilidades de éxito estimadas si el jugador lanza a canasta desde su posición actual, así como las posibilidades de éxito si realiza un pase a otros jugadores.

Mediante esta característica es fácil apreciar si los cálculos de probabilidades se aproximan a la realidad, así como ver como de adecuado es un determinado límite impuesto a los tiros o a los pases.

La información ofrecida en estas etiquetas es fácilmente ampliable si es necesario.

El tercer método consiste en escenarios de prueba. Este método puede utilizarse de dos maneras diferentes.

En primer lugar, podemos utilizarlo para reproducir fácilmente una situación que muestra un problema con la IA. Sin una manera de establecer escenarios de prueba repetibles es muy difícil depurar problemas que puede que se reproduzcan de manera muy poco frecuente en circunstancias muy especiales.

Mediante un escenario de prueba podemos crear la situación concreta que causa el problema para después, probablemente usando el sistema de registros o las etiquetas de depuración, diagnosticar el problema. Una vez resuelta podemos ejecutar de nuevo el mismo escenario de prueba y comprobar que la IA se comporta como esperamos.

El segundo caso de uso es para la realización de posibles mejoras en la IA. Imaginemos que vamos a hacer un cambio a una determinada estrategia de ataque. Podemos ejecutar un escenario de prueba en el que se utilice esa estrategia un alto número de veces y ver cual es el porcentaje de éxito de la jugada. A continuación hacemos nuestro cambio y volvemos a ejecutar el mismo escenario, observando si la probabilidad de éxito ha subido o no.

Los escenarios se definen en ficheros XML y permiten establecer la mayoría de paramétros del partido, como los equipos que juegan, la posición de los jugadores, quien controla la pelota, la IA a utiliar o el estado en el que se encuentran las IAs de ambos equipos.

Para determinar si la prueba ha tenido éxito se utiliza el sistema de eventos. Podemos establecer un evento que determinará si la prueba ha sido un éxito, como por ejemplo, la consecución de una canasta. Así mismo, podemos configurar un evento que hará que el resultado de la prueba sea fallo. Además, el sistema de pruebas también soporta límites de tiempo, de tal manera que una prueba se declare automáticamente éxito o fracaso una vez pasado un determinado tiempo.

A continuación podemos ver un ejemplo de escenario de prueba:

```
<test>
        <match>
                <matchteam home="local" side="left"
                           name="Memphis Grizzlies"/>
                <matchteam home="visitor" side="right"
                           name="Toronto Raptors"/>
        </match>
        <team id="0" ai="test1_0" initialstate="trytoscore">
                <player index="0">
                        <ball/>
                        <pos x="4" y="0"/>
                </player>
        </team>
        <team id="1" ai="test1_1">
                <player index="0">
                        <pos x="6" y="0"/>
                </player>
        </team>
        <success event="5"/>
        <timeout seconds="10" result="failure"/>
</test>
```

En este caso simplemente determinamos las posiciones de un jugador de cada equipo, asignamos la posición de la bola a uno de ellos y decidimos la IA que van a utilizar los equipos. Además, establecemos como estado inicial para el primer equipo *TryToScore*.

El evento número 5 es *Score* por lo que consideramos el test como un éxito si se registra una canasta en los primeros diez segundos y como un fracaso en caso contrario.

Para ejecutar una prueba, invocamos a *inyourface* de la siguiente manera:

```
./inyourface -gametype test -test test-shoot.xml
```

La ejecución del simulador en este caso es totalmente automática y registrará al finalizar el resultado de la prueba.

# Anexo D

# Licencia

A continuación se adjunta el texto de la licencia GNU General Public License, bajo la cual se distribuye el software del presente proyecto.

**GNU GENERAL PUBLIC LICENSE**
Version 2, June 1991



## Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

# Terms and conditions for copying, distribution and modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

(a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

(b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

(a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose

of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. **Because the Program is licensed free of charge, there is no warranty for the Program, to the extent permitted by applicable law. except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Program is with you. Should the Program prove defective, you assume the cost of all necessary servicing, repair or correction.**

12. **In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the**

**program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the Program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.**

**END OF TERMS AND CONDITIONS**

# Índice de figuras

# Índice de tablas

# Bibliografía


[1] Buckland, Mat *Programming Game AI by Example*,
Wordware Publishing, Inc. 2005

[2] *Official Rules of the National Basketball Association*,
http://www.nba.com/analysis/rules_index.html

[3] Reynolds, C.W. *Steering Behaviors For Autonomous Characters*
http://www.red3d.com/cwr/papers/1999/gdc99steer.html

[4] Millington, Ian *Artificial Intelligence for Games*,
Morgan Kauffman Publishers, 2006